# CIRCUITE INTEGRATE DIGITALE

GHEORGHE ŞTEFAN

Editura
DENIX

V 3-el

PACKARD BELL

America grew up listening to us.

It still does.

PACKARD BELL Computers

Sos. PIPERA Nr. 46
BUCURESTI-ROMANIA
Tel.633.50.09 ,312.08.86

# CIRCUITE INTEGRATE DIGITALE

TIPARUL EXECUTAT LA
INTERGRAPH
Bulevardul Păcii nr. 69 A sect 6.
București – România

GHEORGHE ȘTEFAN

# CIRCUITE INTEGRATE DIGITALE

Ministerul Științei și Învățământului
Universitatea Tehnică a Moldovei
Biblioteca Tehnico-Științifică
INVENTAR nr. 972526

Editura DENIX
București, 1993

Această lucrare a fost avizată de colectivul Catedrei de Dispozitive, Circuite şi Aparate Electronice din Facultatea de Electronică şi Telecomunicaţii a Institutului Politehnic Bucureşti.

Copyright © 1993, Editura DENIX
Toate drepturile asupra acestei ediţii sînt rezervate editurii.

ISBN 973-95811-0-2

Redactor: DENISA STEFAN
Tehnoredactare computerizată: Editura DENIX
Coperta: IOANA DRAGOMIRESCU MARDARE
Ilustraţia copertei: prelucrarea pe calculator a desenului *"Banda lui Möbius I"* de *M.C.Escher*.

*Pentru că am găsit, în momente grele, forţa de a definitiva această carte, pentru că, mai apoi, ea a fost publicată cu o grijă pe care numai devotamentul o poate susţine, să-i mulţumim Denisei.*

*Cultura modernă este bolnavă de teorie. Necesitatea de a găsi o formulă distinctă pentru orice situaţie, de a justifica în gînd toate frămîntările realului, a secat energia creatoare şi a răpit omului un anumit simţ al problemelor.*

**Emil Cioran**: "Schimbarea la faţă a României"

*Într-un veac de atîta subtilitate în fizică, matematici, biologie, cum poate fi atîta grosolănie în logică.*

**Constantin Noica:** "Jurnal de idei"

*Că negaţia adevărului e falsitate, încă ar mai merge; dar că negaţia falsităţii e adevăr, Doamne, ce grozăvie!*

**Constantin Noica**: "Jurnal de idei"

# Prefaţă

Obsesia celui ce scrie un curs este dată de căutarea unei maniere în care materialul prezentat să se articuleze. Uneori ai privilegiul ca acest efort să se concretizeze în găsirea unui lucru important, ce depăşeşte statutul, secund, de liant. Căutarea unei ordini în expresie poate să te ducă în faţa unei ordini ce aparţine realităţii pe care încerci să o exprimi cît mai coerent. Aspiraţia către coerenţa expresiei dezvăluie, nu de puţine ori, coerenţe ale realului.

Distincţia combinaţional-secvenţial, devenită tradiţională în domeniul sistemelor digitale, era din ce în ce mai puternic contrazisă de o diversitate ce impunea nuanţări suplimentare. Spectrul funcţional acoperit de abordarea digitală lărgindu-se, s-au produs şi diferenţieri structurale semnificative. Funcţiile de memorare, de control sau cea de procesare apar mult prea diferite ca să mai poată intra, împreună, sub incidenţa aceluiaşi concept: secvenţialul.

Dar nu este vorba numai de încercarea de a clasifica. Trebuie dat socoteală de mecanisme de creştere şi evoluţie, interne domeniului sistemelor digitale. Care este procesul structural care permite accesul către un nivel funcţional superior? Deci, care este *mecanismul de structurare* în domeniul circuitelor şi sistemelor digitale? Acestea sînt întrebările tot timpul prezente în rîndurile, dar mai ales printre rîndurile, acestei cărţi. Întrebarea este perpetuu însoţită de răspunsuri mai mult sau mai puţin directe. Pentru un inginer, ce se pregăteşte în domenii atît de dinamice ca electronica sau calculatoarele, este mult mai important să ştie *ce va deveni* domeniul la care aspiră, decît ceea *ce este* el acum.

Lucrarea prezintă un demers ingineresc, în sensul că încearcă să arate cum se poate depăşi o neputinţă tehnologică dată de o derută teoretică. Spun acest lucru deoarece domeniul sistemelor digitale se află în etapa în care nu poate (tehnologic) să rezolve eficient tot ce-şi propune (teoretic). Nu poate, pentru că încă vrea prea mult şi vrea prea mult pentru că vrea totul, în incapacitatea sa de a preciza ceea ce

într-adevăr este util şi, deci, de dorit. Neintroducînd nici o restricţie în spectrul funcţional pe care-l asociem sistemelor digitale, aspirăm, implicit, către conceperea unor maşini universale. Optimizarea unor structuri cu care dorim să realizăm orice funcţie devine astfel un nonsens.

Materia expusă în acest curs tinde către un maximum, reprezentat de *ideea de procesor*, materializată în microprocesoare sau procesoare microprogramate. Dar este vorba, în primul rînd, de un maxim al ineficienţei, un apogeu care uneşte o gîndire teoretică învechită şi falimentară (prin aspiraţia la maşina universală) cu posibilităţi tehnologice de vîrf. Nu ne rămîne decît să-l parafrazăm pe Elias Canetti spunînd că: "nu se poate scăpa dintr-o dată de un *dispozitiv* (*cuvînt*, la Canetti) periculos. Trebuie mai întîi să te chinuieşti îndelung, aplicîndu-l anapoda". Din acest motiv îl vom studia, pentru început, temeinic.

Încercăm să schiţăm şi evoluţii alternative. Ele sînt prezente pe tot parcursul expunerii, dar mai ales în ultimul capitol, cel destinat reţelelor funcţionale complexe. Concentrarea în spaţiul funcţiilor ce se impun ca utile oferă şansa unei abordări şi soluţionări eficiente. Nu acesta este însă scopul principal al demersului întreprins, care se doreşte numai unul care generează un instrument a cărui performantă utilizare se va putea împlini într-un cadru mai larg, cel al *electronicii funcţionale*.

Ne oprim, de asemenea, la nivelul la care structurile informaţionale apar, dar nu devin esenţiale în specificarea funcţională. Amintim limbajele de asamblare, vorbim despre cele de microprogramare, dar nu abordăm sisteme din categoria calculatoarelor ce se află strict sub incidenţa structurilor informaţionale, autoritar întrupate în programe. Demersul nostru rămîne preponderent structural, în sensul fizic al acestui termen. El va fi completat printr-un curs ce tratează despre "Arhitectura sistemelor de calcul" şi îşi va putea găsi finalizarea contribuind la conturarea cursului de "Electronică funcţională".

Recunoştinţa autorului se îndreaptă către toţi cei ce ştiu şi simt, ei înşişi, că ea trebuie să se manifeste. Este vorba de persoane pentru care listarea propriului nume în finalul unei prefeţe nu reprezintă un fapt important. Prefer, de aceea, să-mi exprim cît mai discret gratitudinea faţă de cei care sincer, dezinteresat şi competent m-au ajutat în diverse etape ale evoluţiei mele profesionale şi umane; lor dedic ceea ce este de luat în seamă din această carte.

G.Ş.

februarie 1992



# Cuprins

**Prefaţă** 7

**Cuprins** 9

**Cuprins extins** 11

**1. INTRODUCERE** 19

**2. STRUCTURAREA SISTEMELOR DIGITALE** 25

**3. CIRCUITE LOGICE COMBINAŢIONALE**

**Sisteme de ordinul zero** 31

3.1 Descrierea formală a circuitelor logice combinaţionale 31
3.2 Implementarea cu porţi a circuitelor combinaţionale 50
3.3 Utilizarea circuitelor standard în implementare 62
3.4 Structuri programabile 72
3.5 Circuite specializate 83
3.6 Complexitatea circuitelor combinaţionale 90

**4. CIRCUITE DE MEMORARE** 97

**Sisteme de ordinul unu** 98

4.1 Efectul unei bucle de reacţie peste un circuit logic combinaţional 101
4.2 Latch-uri 107
4.3 Memorii 124
4.4 Structuri master-slave 129
4.5 Registre 130
4.6 Registrul de stare 132
4.7 Memoria FIFO

5. AUTOMATE FINITE ELEMENTARE 135
Sisteme de ordinul doi 136
5.1 Structura de principiu a unui automat finit 143
5.2 Automate bistabile 151
5.3 Numărătoare 166
5.4 Descrierea formală a automatelor finite 181
5.5 Implementarea automatelor finite 204
5.6 Automate cu reacţie negativă 220
5.7 Automate cu spaţiul stărilor structurat 230
5.8 Automate orientate funcţional 238
5.9 Complexitatea combinaţională a automatelor finite

6. PROCESOARE 249
Sisteme de ordinul trei 250
6.1 Automate ca sisteme de ordinul trei 266
6.2 Automate cu memorie 270
6.3 Automate cu stivă 272
6.4 Procesorul elementar 283
6.5 Procesorul 301
6.6 Microprocesorul 331
6.7 Procesoare microprogramate 345
6.8 Concluzii

7. CONCLUZII 347

8. REŢELE FUNCŢIONALE COMPLEXE 351
8.1 Ce este un sistem complex? 352
8.2 Reţele combinaţionale 353
8.3 Reţele cu memorie 367
8.4 Reţele de automate 374
8.5 Comentarii finale 379

9. ANEXE 383
9.1 Notaţii neexplicate în text 383
9.2 Parametrii porţilor logice elementare 384
9.3 Porţi elementare 387

10. COMENTARII BIBLIOGRAFICE 393

Abstract 399

Extended contents 401



# Cuprins extins

1. **INTRODUCERE** 19
Continuu-discontinuu. Semnal-simbol. Combinaţional-secvenţial. Secvenţial-paralel. Maşini universale-structuri funcţionale. Formal-neformalizat-neformal.

2. **STRUCTURAREA SISTEMELOR DIGITALE** 25
Extensia serie. Extensia paralel. Extensia serie-paralel. Introducerea unei bucle de reacţie. Clasificarea în ordine.

3. **CIRCUITE LOGICE COMBINAŢIONALE**
**Sisteme de ordinul zero** 31
3.1 Descrierea formală a circuitelor logice combinaţionale 31
3.1.1 Variabile şi funcţii logice 31
3.1.2 Conectivi logici 33
3.1.3 Formalizarea algebrei logice 36
3.1.4 Minimizarea funcţiilor logice 40
Procedee algebrice. Diagrame Veitch-Karnaugh. Funcţii incomplet definite. Diagrame cu variabile incluse.
3.1.5 Funcţii definite în limbaj natural 48
3.2 Implementarea cu porţi a circuitelor combinaţionale 50
3.2.1 Circuite combinaţionale cu o ieşire 50
Timp de propagare. Fan-out şi fan-in. Utilizarea porţilor inversoare. Complexitatea şi adîncimea circuitelor combinaţionale. Hazardul. Evitarea hazardului.
3.2.2 Circuite combinaţionale cu mai multe ieşiri 59

Sumatorul complet. Minimizare globală. Optimizarea adîncimii. 62
3.3 Utilizarea circuitelor standard în implementare 62
3.3.1 Circuite implementate cu multiplexoare
MUX cu valori pe intrare. MUX cu variabile pe intrare. MUX cu funcţii pe intrare. Comparaţie. 65
3.3.2 Circuite implementate cu demultiplexoare
Oportunitate. Folosirea NAND-urilor sau a AND-urilor. Circuitul de exponenţiere.
3.3.3 Utilizarea multiplexoarelor şi demultiplexoarelor în implementarea circuitelor combinaţionale 68
Exemplu. Calculul complexităţii maxime.
3.3.4 Extinderea multiplexoarelor şi demultiplexoarelor 70
3.4 Structuri programabile 72
3.4.1 Memorii fixe (ROM) 73
Definiţie. Adresarea bidimensională. Exemplu. Extinderea capacităţii ROM-urilor. Utilizarea eficientă.
3.4.2 Matrici logice programabile (PLA) 79
Definire prin limitare. Exemplu. Alternativa logic-aritmetic. Extinderea capacităţii. PGA. Arii de porţi.
3.5 Circuite specializate 83
3.5.1 Circuite cu funcţii dedicate 83
Sumator. Comparator. Codificator cu prioritate.
3.5.2 Circuite cu funcţii selectabile 84
Cod de selecţie optimal. ALU.
3.5.3 Circuite combinaţionale de deplasare 87
Principiul Barrel shifter. Complexitate şi adîncime.
3.6 Complexitatea circuitelor combinaţionale 90
Notaţii pentru magnitudine. Formulă-circuit.
3.6.1 Circuit logic Universal 92
3.6.2 Teorema lui Spira 93
Exemple. Demonstrarea teoremei. Aplicarea teoremei.

**4. CIRCUITE DE MEMORARE**

**Sisteme de ordinul unu** 97
4.1 Efectul buclei de reacţie peste un circuit logic combinaţional 98
4.1.1 Structuri oscilante 98
Exemplu. Amorsarea oscilaţiei. Stabilitatea parametrilor.
4.1.2 Structuri care se blochează 99
Structuri perfect simetrice. Structura parţial asimetrică. Structură complet asimetrică. "Experiment" ce probează imposibilitatea identităţii.
4.2 Latch-uri
4.2.1 Latch-ul elementar 101
Segregarea funcţiei de latch. Structura latch-ului. Deficienţe. 101



4.2.2 Latch-ul cu ceas 104
Structură. Limite. Timpii de set-up şi hold.
4.2.3 Latch-ul de tip D (date) 106
Restricţionare. Aplicaţii.
4.3 Memorii 107
4.3.1 Latch-ul adresabil 107
Schemă. Aplicaţie. Limitări.
4.3.2 Structura de principiu a unei memorii cu acces aleator (RAM) 109
Structură liniară. MUX bidirecţional. Celula elementară. RAM-ul ca reţea bidimensională.
4.3.3 Memoria dinamică - DRAM 113
Celula de memorie dinamică. Celula cu un singur tranzistor. Structura de multiplexor/demultiplexor a memoriei dinamice. Adresarea unui DRAM. Cipul DRAM. Timingul.
4.3.4 Matrice de cipuri DRAM 120
Schema de principiu a unei memorii. Timing. Generatorul de ciclu.
4.4 Structuri master-slave 124
4.4.1 Principiul master-slave 124
Schemă de principiu. Timpi caracteristici. Relaţia dintre $t_H$ şi $t_p$. Întîrzierea secvenţială.
4.4.2 Registrul de deplasare serie 127
4.5 Registre 129
4.5.1 Registrul paralel 129
4.5.2 Registrul serie-paralel 129
4.6 Registrul de stare 130
Definiţie. Tabel de condiţionare. Schemă bloc.
4.7 Memoria FIFO 132
Funcţie. Structură. Aplicaţie.

**5. AUTOMATE FINITE ELEMENTARE**

**Sisteme de ordinul doi** 135
5.1 Structura de principiu a unui automat finit 136
5.1.1 Segregarea funcţională 136
Exemplu. Prelucrare/Memorare.
5.1.2 Automate asincrone - automate sincrone 139
5.1.3 Structurarea neechilibrată a automatelor sincrone 142
5.2 Automate bistabile 143
5.2.1 Bistabilul T 143
Deducerea funcţiei minimale pentru un automat. Calculul buclei. Divizarea frecvenţei. Numărare modulo doi. Transformare în bistabil D. Exemplu de structurare excesivă.
5.2.2 Bistabilul J-K 146

Funcţia automatului bistabil cu două intrări. Calculul buclei. Autonomia. Transformarea în bistabili de tip T şi D. Semitransparenţa bistabilului JK. Bistabilul JK activat exclusiv pe tranziţia ceasului.
5.2.3 Structura optimă a bistabilului de tip D 150
5.3 Numărătoare (automate $2^n$-stabile) 151
5.3.1 Numărătoare asincrone 152
Divizarea frecvenţei cu $2^n$. Structura numărătorului. Numărarea inversă. Timpul de numărare. Asincronismul. Complexitatea.
5.3.2 Numărătoare sincrone 154
Principiul. Cascadarea automatelor bistabile. Timpul de numărare: O(1) sau O(n). Complexitatea. Variantă cu timp de numărare O(log n). Numărătoare reversibile. Numărătoare up-down. Numărătoare presetabile.
5.3.3 Divizoare programabile 159
Schema de principiu. Realizarea cu numărătoare presetabile asincron sau sincron. Hazardul la ieşirea divizată.
5.3.4 Structură de memorie FIFO 161
Structura abstractă. Implementare. Complexitate.
5.3.5 Numărătoare Moebius 164
Definiţie. Implementare. Amorsarea funcţionării. Evitarea blocării. Aplicaţii.
5.4 Descrierea formală a automatelor finite 166
5.4.1 Definiţii. Proprietăţi 167
5.4.2 Reprezentarea automatelor 170
Grafuri. Tabele de tranziţie. Organigrame. Tabele de adevăr. Diagrame de tranziţie. Pseudocod. Forme de undă. Limbajul natural.
5.4.3 Minimizarea automatelor
5.5 Implementarea automatelor finite 180
5.5.1 Codificarea stărilor 181
Izomorfismul în spaţiul stărilor. Dependenţa redusă. Variaţia minimă. Asincronismul intrărilor. Hazardul la ieşire. Codificări combinate. Coduri nefolosite. 181
5.5.2 Limite fundamentale 185
Imposibilitatea testării mai multor variabile asincrone în aceeaşi stare. Mealy imediat fără hazard. Mealy cu întîrziere cu variabile de intrare asincrone. Moore imediat cu intrări asincrone şi fără hazard.
5.5.3 Exemple de proiectare 187
Variante de codificare a stărilor la un semiautomat. Automat Moore cu intrări asincrone. Topologie de organigramă ce nu permite codare cu dependenţă minimă. Automat Mealy şi automatul Moore echivalent. Moore fără hazard. Topologie de organigramă ce nu permite



codificarea cu variaţie minimă. Exemplu de Moore imediat cu variabile asincrone şi fără hazard. Soluţii cu introducerea stărilor suplimentare. Translaţia Moore-Mealy. Frecvenţa maximă a ceasului.
5.6 Automate cu reacţie negativă 204
5.6.1 Definiţie. Exemple 204
Exemplu limită. Reacţie negativă maximă. Structură limită asociată reacţiei negative maxime.
5.6.2 CROM 210
Alternativa comandă-control. Structură fundamentală. Variante optimizate. CROM-ul ca Moore imediat. Linia de comandă. Exemplu. Translatarea binară a liniei de comandă. ROM-PLA.
5.7 Automate cu spaţiul stărilor structurat ($AS^3$) 220
5.7.1 Definiţie. Efecte 220
Complexitatea $AS^3$. Raportarea la reacţia negativă. Accentuarea caracterului secvenţial al evoluţiei.
5.7.2 Configurarea semiautomatelor cu spaţiul stărilor structurat 222
Structura detaliată. Structură cu latch master partajat. Structură cu RAM.
5.7.3 RALU 225
RAM multiport. Structura RALU. Variantă pentru echilibrarea buclei. Specificarea funcţională. Exemplu.
5.8 Automate orientate funcţional 230
5.8.1 Acumulator de produse 231
5.8.2 Memorie de tip stivă (LIFO) 232
Definiţie. Implementare. Variantă. Ordinul structural.
5.8.3 Memoria conexă 235
Definiţie. Exemplu de funcţionare. Automat cu memorie conexă.
5.9 Complexitatea combinaţională a automatelor finite 238
5.9.1 Conversia automat finit - circuit combinaţional 239
5.9.2 Echivalentul combinaţional al unui semiautomat, cu $D(n) = O(\log n)$ 241
Exemplu de aplicare a teoremei lui Spira. Complexitatea circuitului tip *formulă*. Complexitatea reţelei tip *circuit*.
5.9.3 Structură combinaţională generală asociată unui semiautomat finit 246
Echivalarea cu o reţea de multiplexoare. Introducerea reacţiei negative maxime.

**6. PROCESOARE**
**Sisteme de ordinul trei** 249
6.1 Automate ca sisteme de ordinul trei 250
6.1.1 Automate realizate cu "registre" de J-K-uri 251

Regula de transformare a diagramei de tranziţie. Metoda de implementare. Exemple. De ce se reduce dimensiunea CLC-ului? De ce nu apare efectul şi la bistabile T? Calculul perioadei minime a ceasului.
6.1.2 Automate realizate cu numărătoare sincron presetabile 259
Principiul de realizare. Exemplu. Restricţii. Generator de ciclu pentru o memorie DRAM.
6.2 Automate cu memorie 266
RALU cu bistabil de CR. Creşterea autonomiei. Determinarea maximului într-un şir. Segregarea în spaţiul stărilor.
6.3 Automatul cu stivă 270
Definiţie. Exemplu: recunoaşterea apartenenţei la $\{a^n b^n \mid n \geq 1\}$.
6.4 Procesorul elementar 272
6.4.1 Principiul de structurare 272
Definiţie. Structură. Aplicaţie: media pe ultimele două elemente recepţionate. Variante: CROM cu increment, CROM cu testarea condiţiei negate. Reluarea aplicaţiei.
6.4.2 Conectarea procesorului elementar 279
Directă, între două sisteme. Prin FIFO. La LIFO. Conectare bidirecţională la RAM. PE este o structură orientată funcţional.
6.5 Procesorul 283
6.5.1 Procesoare fără bus de adresă 284
Procesor cuplat cu două FIFO. Exemplu: procesor aritmetic. Procesor orientat pe stivă. Exemplu: evaluarea de expresii. Limite.
6.5.2 Procesorul cu bus de adresă 292
Structură. Limbajul LOOP. Limbajul maşină. Funcţionarea procesorului. Variante ale limbajului maşină.
6.6 Microprocesorul 301
6.6.1 Arhitectura microprocesorului 302
Ideea de arhitectură. Sistemul în care este interconectat $\mu$P. Funcţia RAM-ului. Schema de principiu. Limbajul maşină (setul de instrucţiuni).
6.6.2 Funcţionarea microprocesorului 306
Formatul instrucţiunilor. Registrul de stare. Structura internă. Descrierea în pseudocod. Descrierea prin organigramă. Registru pentru program counter.
6.6.3 Limbajul de asamblare 320
Cod binar. Cod octal/haxazecimal. Cod simbolic. Limbaj de asamblare. Limbaj de nivel înalt.
Asamblor/Compilator. Exemple.
6.6.4 Logica programată 324
Definiţie. Exemplu: generator de "bară". Sistem cu histerezis.



6.6.5 Concluzii 330
6.7 Procesoare microprogramate 331
6.7.1 Limbaj de microprogramare 332
Definiţie. Microrutine de bază. Exemplu: sumă.
6.7.2 Variante structurale ale procesoarelor microprogramate 337
Testarea condiţiei negate. Registru pipe-line. Implicaţii ale registrului pipe-line. CROM cu LIFO.
6.7.3 Microprogramarea dinamică 342
Definiţie. Necesitatea microprogramării dinamice. Mecanisme de încărcare a SRAM-ului. Bazarea.
6.7.4 Concluzii 345
6.8 Concluzii 345

**7. CONCLUZII** 347
Corelaţia funcţie - nivel structural. Sisteme de ordinul patru. Structurarea neechilibrată.

**8. REŢELE FUNCŢIONALE COMPLEXE** 351
8.1 Ce este un sistem complex? 352
8.2 Reţele combinaţionale 353
8.2.1 Reţele de calcul al prefixelor (RCP) 354
Definiţie. Reţea cu $D(n) = O(\log n)$. Reţea definită recursiv. Circuit de incrementare.
8.2.2 Reţele de sortare 359
Definiţie. Exemplu. Principiul zero-unu. Sortatorul Batcher.
8.2.3 Reţele de permutare 364
Definiţie. Permutatorul Waksmann. Aplicaţii.
8.3 Reţele cu memorie 367
8.3.1 Reţele pipe-line 367
Necesitate. Exemplu. Timpul de prelucrare.
8.3.2 Reţele sistolice de ordinul unu 371
Elementul reţelei. Sistem de înmulţire a unui vector cu o matrice.
8.4 Reţele de automate 374
8.4.1 RCP ca reţea de automate 374
8.4.2 Sortator sistolic 376
Sortator elementar. Reţeaua de sortare. Exemplu de funcţionare. Structurarea echilibrată a sortatorului.
8.5 Comentarii finale 379
Migrarea funcţională. Aria VLSI. De ce numai $D(n)=O(\log n)$? Presiunea secvenţialului.

**9. ANEXE** 383
9.1 Notaţii neexplicate în text 383
9.2 Parametrii porţilor logice elementare 384
9.2.1 Caracteristica de transfer 384

Ministerul Ştiinţei şi Învăţământului
Universit[illegible] Tehnică a Moldovei
Biblioteca Tehnico-Ştiinţifică
INVENTAR nr. 972526

9.2.2 Timpii de comutaţie 385
9.2.3 Marginea de zgomot 387
9.3 Porţi elementare 387
9.3.1 Circuite CMOS 387
9.3.2 Circuite TTL 389
**10. COMENTARII BIBLIOGRAFICE** 393



1

# INTRODUCERE

O posibilă delimitare netă în domeniul sistemelor electronice o constituie cea dată de alternativa *continuu-discontinuu* referitor la semnalele ce pot fi aplicate la intrare sau pot fi recepţionate la ieşirea circuitelor. La cele două limite avem semnalul pur sinusoidal

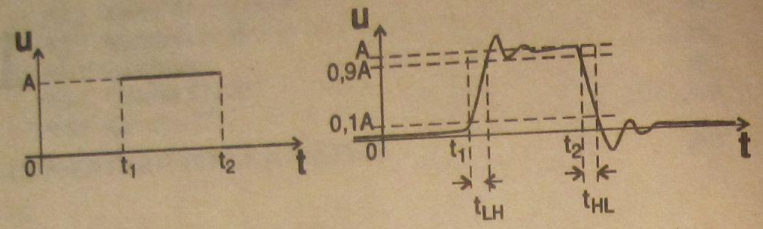
$$s(t)=A\sin\omega t, \text{ pentru } t \in (-\infty,+\infty)$$

şi impulsul dreptunghiular

$$u(t)=\begin{cases} A & \text{pentru } t \in [t_1, t_2] \\ 0 & \text{pentru } t \notin [t_1, t_2] \end{cases}$$

Ambele semnale sînt la fel de ireale. Primul presupune absenţa absolută a unei discontinuităţi iar cel de-al doilea defineşte tranziţii în timp nul. Zeroul şi infinitul fiind în egală măsură inabordabile inginereşte (dacă nu chiar complet inabordabile), aceste semnale nu sînt practic realizabile. Continuul şi discontinuul sînt concepte formale, utile pentru delimitări teoretice care, practic, au sens prin tehnicile distincte pe care le presupun, dar nu prin ceea ce în esenţă ar vrea să semnifice.

Pornind de la cel de-al doilea tip de semnal se poate defini o *electronică a discontinuului* care să opereze cu semnale oricît de complexe obţinute pornind de la *impulsul dreptunghiular*. În figura 1.1*a* este reprezentată funcţia $u(t)$, care aproximează un semnal real ce uneori poate lua forma celui din figura 1.1*b*. Realitatea impune tranziţii în intervale de timp diferite de zero ($t_{LH}$,$t_{HL}$) iar atingerea *nivelurilor* de *zero* sau $A$ nu se poate face net ci prin aproximări succesive ce se stabilizează în intervale de timp, uneori, supărător de mari. Unde începe




a) b)

**Figura 1.1**

tranziţia semnalului? Pentru că nu se poate stabili cu rigurozitate se va considera, arbitrar, că tranziţia pozitivă începe cînd mărimea semnalului este *0,1A*. Similar se judecă pentru celelalte discontinuităţi. Mărimea supracreşterilor se exprimă procentual faţă de *A*. Timpul de creştere între *0,1A* şi *0,9A* este denumit timpul de *front* pozitiv sau, simplu, front pozitiv, notat cu $t_{LH}$. Similar este definit frontul negativ, notat cu $t_{HL}$.

Discontinuitatea semnalului se manifestă atît în *timp* cît şi în *mărime*.

Timpul este marcat discontinuu printr-un semnal periodic dreptunghiular denumit, de regulă *ceas* şi notat prescurtat prin *CK* (figura 1.2). Acest semnal este caracterizat prin *perioada* de repetiţie $T$, prin *factorul de umplere*, $t_1/T$ şi prin *frontul activ* (pozitiv în reprezentarea noastră). Orice eveniment într-un *sistem discret sincron* este o consecinţă a frontului activ al semnalului de ceas. Dacă într-un sistem sau o parte a lui nu este aplicat un *CK* comutările sînt *asincrone* şi ansamblul va fi considerat *asincron*. Timpul fiind marcat de frontul activ al *CK*, este asigurată discretizarea scurgerii acestuia. Stările sistemului vor putea fi luate în consideraţie numai la aceste intervale de timp net distincte.

Mărimile semnificative ale semnalului vor putea fi numai *zero* şi *A*, cele intermediare fiind cu obstinaţie evitate. Această maximă discretizare în mărime a semnalelor va permite realizarea unor funcţii de transfer cu acurateţe strict controlabilă. *Zero* şi *A* sînt mărimi ideal definite. În realitate se iau în consideraţie un nivel maxim pentru *zero* şi unul minim pentru *A*. Cele două valori pe care le poate lua *u(t)* permit realizarea unor *circuite foarte simple* care să funcţioneze cu *acurateţe maximă* în *structuri de foarte mare complexitate*.

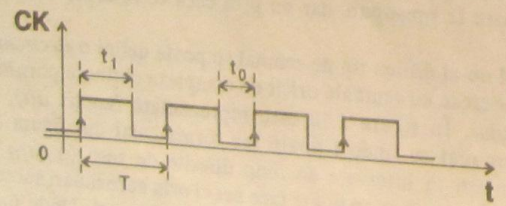



**Figura 1.2**



• O caracteristică esenţială a sistemelor discrete este aceea că în definirea funcţională acceptă **corespondenţa semnal-simbol** (simbol în sensul său tehnic, de mărime cu semnificaţie). Astfel, putem să asociem celor două niveluri ale semnalului *u(t)* şi a altora similare *valori logice sau numerice*.

În *logică pozitivă* vom asocia *nivelului zero* al semnalului *u(t)* semnificaţia logică de *fals*, iar *nivelului A*, semnificaţia logică de *adevărat*. Dacă, aceste *valori logice* sînt alocate invers (adevărat cu *zero* şi fals cu *A*), atunci spunem că lucrăm în *logică negativă*.

Obişnuim să asociem nivelul *zero* cu numărul binar *0* şi nivelul *A* cu numărul binar *1*. Spunem, spre exemplu, că *u(t)* comută, la momentul $t_1$, din *0* în *1*, sau din *zero logic* ("0") în *unu logic* ("1"), această din urmă exprimare introducînd o redundanţă, de multe ori necesară.

Circuitele ce prelucrează astfel de semnale vor fi denumite, de regulă *circuite logice* sau *digitale* (numerice) pornind de la aceste corespondenţe semnal-simbol. Vom nuanţa însă uneori semnificativ între o *interpretare* logică a funcţiei unui circuit şi una numerică.

• **Combinaţional - secvenţial**: o alternativă clasică în domeniul sistemelor digitale. Pentru a o explicita sînt necesare precizări iniţiale sub forma definiţiilor ce urmează.

**Definitia 1.1** O *mulţime de configuraţii binare* este

$$A = \{A_1, A_2, ..., A_n\},$$

unde $A_k = a_m a_{m-1}...a_0$, pentru $k = 1,...,n$, cu $a_i \in \{0, 1\}$, pentru $i = 0, 1,..., m$, fiind îndeplinită condiţia

$$n \le 2^{m+1}. \square$$

Configuraţia binară $A_k$ este formată din $m+1$ *biţi* deoarece $a_i$ nu ia decît *două* valori (logice sau numerice) şi în consecinţă, fiecare simbol de tip $a_i$ conţine (după Shannon) informaţie de *un bit*.

**Definiţia 1.2** Dacă

$$n = 2^{m+1},$$

atunci configuraţia binară este *complet definită*, altfel este incomplet definită. □

Vom întîlni de multe ori probleme în care nu sîntem obligaţi să luăm în consideraţie toate cele $2^p$ configuraţii binare de $p$ biţi.

**Definiţia 1.3** Un *sistem logic (digital)* este definit prin

$$S = (X, Y, f)$$

unde $X$ este mulţimea de configuraţii binare complet definită a valorilor de *intrare*, $Y$ este mulţimea configuraţiilor binare de *ieşire*, iar $f$ este funcţia de transfer de forma

$$f: X \to P^*(Y).$$

(Prin $P^*(Y)$ am notat mulţimea părţilor nevide ale lui $Y$.) □

În funcţie de momentul de timp (tactul) în care se aplică $X_i \in X$ la intrarea unui sistem $S$, se pot obţine valori distincte pentru ieşire. Ieşirea este deci determinată şi ca o consecinţă a unei variabile interne a sistemului ce nu a fost încă pusă în evidenţă. Vom numi această variabilă *stare internă* sau simplu *stare*. Ieşirea va fi, deci, impusă în funcţie de valoarea $X_i$ a variabilei de intrare şi valoarea pe care o ia variabila de stare internă, putînd lua valori în mulţimea $P_i(Y) \in P^*(Y)$.

**Definiţia 1.4** Atunci cînd cardinalul mulţimilor $P_i(Y) \in P^*(Y)$ este egal cu unu pentru orice $i$, sistemul logic este un *sistem (circuit) logic combinaţional* notat prescurtat prin *CLC*. □

Funcţia de transfer a unui *CLC* este $f_c : X \to Y$, fiecărei configuraţii binare de intrare corespunzîndu-i una de ieşire. Ieşirea rezultă printr-o *combinaţie* oarecare realizată pornind *numai* de la biţii aplicaţi pe intrare.

**Definiţia 1.5** Dacă un sistem logic nu este *CLC*, atunci este *secvenţial*. □

Un circuit este secvenţial dacă ieşirea lui depinde de ordinea în care sînt aplicate la intrare configuraţiile binare, astfel încît pentru aceeaşi configuraţie binară la intrare, poate răspunde diferit pe ieşire în funcţie de secvenţa anterior aplicată intrării, care poate aduce *starea internă* a sistemului la valori diferite.

**Exemplul 1.1** Calculatorul de buzunar este un circuit logic secvenţial. Într-adevăr, dacă tastăm *2*, *+*, *5*, *=*, afişează *7* iar dacă tastăm *2*, *+*, *6*, *=*, la ieşire (pe sistemul de afişaj) obţinem *8*. Acţiunea tastei = a fost diferită în cele două cazuri în funcţie de secvenţa tastărilor (intrărilor) anterioare. Acest sistem "*ţine minte*", parţial măcar, ceea ce s-a întîmplat cu el în trecutul mai îndepărtat sau mai apropiat, pentru ca în cazul unor comenzi particulare (=, în exemplul nostru) să reacţioneze în funcţie de ceea ce a *memorat*. O caracteristică esenţială este aceea că posedă facilităţi de memorie materializate în posibilitatea de a avea o stare internă.

Diversitatea funcţională pe care au căpătat-o sistemele digitale nu mai permite clasificarea clasică, rudimentară, în *CLC* şi *circuite secvenţiale*, această din urmă categorie nuanţîndu-se foarte mult. În următorul capitol vom evidenţia criterii ce duc la o ordonare mai consistentă cu realitatea curentă.

• **Secvenţial - paralel** este o alternativă ce marchează puternic dezvoltarea domeniului sistemelor digitale în ultimile două decenii, chiar dacă nu-i este specifică. Pentru a săpa, spre exemplu, un canal de irigaţii putem pune în funcţiune *în paralel* mai multe excavatoare, echidistant distribuite pe traseul acestuia, pe cînd pentru a săpa un puţ de mină nu putem folosi decît un singur utilaj care va lucra avansînd pas cu pas (*secvenţial*) de la suprafaţă spre adîncime. În primul caz scopul urmărit permite declanşarea unor *procese paralele* pe cînd în cel de-al doilea sîntem obligaţi la o abordare *secvenţială*, care ar fi fost posibilă şi în primul cu o penalizare corespunzătoare în *timpul de execuţie*.

Un alt exemplu, mai apropiat de domeniul nostru, îl constituie evaluarea expresiei

$$E_1 = a_1 b_1 + a_2 b_2 + a_3 b_3 + a_4 b_4$$

pentru un set de valori ale variabilelor $a_i$, $b_i$. *Secvenţial*, evaluarea presupune şapte paşi (patru produse şi trei sume), pe cînd paralel, calculul poate fi făcut în trei paşi, dacă operaţiile pe care le pot face maşinile sînt numai operaţii binare (cu doi operanzi). Astfel, în *primul pas*, patru maşini efectuează produsele $a_i b_i$ pentru



$i = 1, ..., 4$, în cel de *al doilea pas* se sumează primele două produse cu o maşină şi ultimele două cu altă maşină, iar în *ultimul pas* o maşină sumează rezultatele obţinute în cel de al doilea pas. Dacă luăm în consideraţie expresia

$$E_2 = (...(((a_1 + a_2)a_3 + a_4)a_5 + a_6)...)$$

şi dorim să o evaluăm aşa cum este scrisă nu vom putea folosi operandul $a_3$ înainte de efectuarea sumei $(a_1 + a_2)$. La fel $a_4$ nu poate fi sumat înainte de evaluarea subexpresiei $(a_1 + a_2)a_3$, ş.a.m.d.

Pentru evaluarea expresiei $E_1$ au putut fi declanşate *procese paralele*, într-o maşină adecvată, pe cînd pentru $E_2$, oricît ar fi de complexă maşina, nu poate fi efectuată mai mult de *o operaţie* la un moment dat. $E_2$ nu admite procese paralele de evaluare, impunînd o evaluare secvenţială.

Paralelismul permite ca printr-un *efort structural* (mai multe maşini ce funcţionează corelat) să creştem, cel mult proporţional, *viteza de evaluare*, de răspuns, a unui sistem digital. Dar aşa cum zece muncitori sapă mai repede un şanţ dar nu şi o gaură pentru un stîlp de telegraf, tot aşa nu orice problemă îşi va putea găsi o soluţie mai rapidă şi mai eficientă folosind mai multe sisteme digitale, oricît de adecvat interconectate.

Paralelismul este o alternativă ce va trebui luată în consideraţie la toate nivelurile structurale ce vor fi abordate. Vor fi însă multe situaţii în care caracterul inerent secvenţial al evaluării nu va putea fi evitat.

• **Maşini universale - structuri funcţionale.** Sistemele digitale pot fi gîndite în două moduri. O primă variantă presupune structuri care execută o funcţie bine precizată. Primind la intrare un şir de simboluri codificate binar, se obţine la ieşire un răspuns ce corespunde unei anumite funcţii de transfer. Funcţia este strict ataşată structurii care primeşte la intrare *numai* şirul de prelucrat. A doua variantă admite la intrare şi un şir prin care-i este definită funcţia. Este vorba de o structură ce potenţial poate realiza *orice* funcţie. Actualizarea funcţională se face printr-un şir de simboluri numit, de regulă, *program*.

În primul caz avem de a face cu *structuri funcţionale* iar în cel de al doilea cu *maşini universale*. Primele permit o foarte bună adecvare la scopul urmărit, atît în ceea ce priveşte complexitatea structurală cît şi performanţele de viteză. Maşinile universale posedă o maximă flexibilitate funcţională care se poate adapta unor probleme concrete în condiţii în care nu întotdeauna se pot atinge performanţe maxime. Universalitatea se plăteşte prin creşterea complexităţii şi limitarea vitezei în execuţie.

Evoluţia tehnologiilor repune tot timpul în discuţie alternativa prezentată. La fiecare aproximativ *10* ani se repune în discuţie oportunitatea soluţiilor oferite de cele două puncte de vedere. Etapa prin care trecem tinde să impună structurile funcţionale datorită performanţelor şi flexibilităţilor tehnologice ce cresc continuu.

Este foarte tentant să ne imaginăm că se vor putea rezolva performant toate problemele cu o unică structură actualizabilă funcţional prin programare, dar pare la fel de spectaculos să gîndim tehnologii de proiectare şi realizare care să ne permită obţinerea eficientă a unor structuri deosebit de performante pentru *orice* funcţie formal definibilă. Din aceste motive nici una din cele două tendinţe nu

trebuie abandonată deoarece, credem că, jocul tehnologiilor ni le vor impune alternativ în atenţie.

• **Formal - neformalizat - neformal.** Pentru analiza şi sinteza sistemelor digitale dispunem de formalisme adecvate care pot fi utilizate mai simplu sau mai complicat în proiectare. Algebra logică, teoria automatelor, tehnicile de programare sau microprogramarea, teoria algoritmilor, a complexităţii calculului, şi altele ne stau la dispoziţie ca instrumente *formale* ce permit stăpînirea domeniului şi oferă inginerului posibilităţi deosebite de descriere, analiză sau proiectare. Orice funcţie ce poate fi descrisă formal îşi poate găsi o soluţie teoretică iar practic foarte multe din ele sînt deja realizabile. Din păcate există multe funcţiuni pentru care nu posedăm descrieri formale, motiv pentru care soluţionarea lor nu se întrevede încă. Limitele formalului sînt şi limitele teoretice ale realizabilului în domeniul sistemelor digitale. Practic limitele formalului sînt în momentul de faţă de neatins cu sisteme concrete. Sub incidenţa *neformalului* rămîn încă funcţiuni importante ce par înrudite cu cele pe care deja le realizăm. Pentru mulţi dintre noi, dacă un sistem digital poate realiza calcule complicate, atunci pare firească aspiraţia ca, prin perfecţionări accesibile, un astfel de sistem să devină inteligent. Tranziţia nu este aşa de simplă deoarece pentru calcul există un model formal avansat, pe cînd inteligenţa, prin ceea ce are ea esenţial, este un proces pe care nu l-am formalizat şi se pare că nu acceptă o formalizare în sensul clasic. Între inteligenţa artificială a calculatoarelor şi cea naturală a vieţuitoarelor există diferenţa dată cel puţin de distincţia dintre formalul, tehnic realizabil, şi neformalul pentru care soluţiile tehnice, chiar dacă sînt uneori posibile sub forma biotehnologiilor, nu constituie încă o garanţie a realizabilităţii.

Un sistem digital va putea imita destul de bine unele comportamente inteligente dar nu inteligenţa. Va rămîne întotdeauna un rest în care se va găsi esenţialul. Un obiect tehnic poate să ne mulţumească şi dacă acoperă, într-un domeniu limitat, majoritatea comportamentelor previzibile. În acest sens descrierea formală, prin procedeele inteligenţei artificiale, spre exemplu, poate fi practic utilă dacă precizăm nişte restricţii şi acceptăm nişte limite imprevizibile.

Limitele descrierilor formale şi ale realizărilor practice ce derivă din ele pot fi întrevăzute chiar din natura semnalelor cu care operează sistemele digitale. Operînd cu distincţia netă dintre *adevărat* şi *fals*, dintre *1* şi *0*, nu se poate nuanţa, în sisteme oricît de complexe, astfel încît să se surprindă naturalul în toate comportamentele sale. Realitatea scapă unui formalism care este atît de tranşant în fundamentele sale. Complexitatea pe care o pot atinge sistemele digitale nu va putea compensa niciodată simplificările fundamentale pe care se bazează.

Eludarea formalului este uneori folosită pentru a realiza simplificări notabile în procesul de analiză sau sinteză. Procedeele formale pot fi uneori ignorate impunîndu-se soluţii directe, *neformalizate*, sugerate de o înţelegere directă sau de o experienţă anterioară bine asimilată. Nu vom ocoli niciodată astfel de căi, mai ales atunci cînd putem evita formalisme laborioase. Întotdeauna va fi însă vorba de probleme care comportă şi o soluţie formală.

Acest domeniu chiar dacă este bine formalizat aspiră către rezolvarea unor probleme neformale şi acceptă soluţii prezentate neformalizat strict. Ingineria îşi atinge astfel, limite spectaculoase în domeniul sistemelor digitale.



# 2

# STRUCTURAREA SISTEMELOR DIGITALE

Abordarea clasică în domeniul sistemelor digitale a impus o clasificare simplă a acestora în *circuite combinaţionale* şi *circuite secvenţiale*. Creşterea complexităţii funcţionale a atras dupa sine nuanţări care presupun evidenţierea unui *principiu de creştere structurală* în virtutea căruia vom obţine o clasificare mai consistentă.

Avînd la dispoziţie sistemele $S_i = (X_i, Y_i, f_i)$, pentru $i = 1,..., n$, cum se pot obţine sisteme mai complexe? De asemenea, care sînt modalităţile prin care putem delimita, în clasa sistemelor compuse, categorii distincte, care să permită o ordonare a acestora? Vom pune în evidenţă, pentru a răspunde la aceste întrebări, patru modalităţi de interconectare a sistemelor: interconectarea *serie*, *paralel*, *serie-paralel* şi conectarea în *buclă*.

• **Extensia serie** presupune o interconectare de tipul celei din figura 2.1 în care două sisteme digitale, $S_1$ şi $S_2$, sînt conectate astfel încît ieşirea primului este aplicată celui de al doilea. Se obţine sistemul

$$S = (X_1, Y_2, f)$$

unde

$$f: X_1 \to P^*(Y_2)$$

este obţinută prin compunerea

$$f = f_1 \circ f_2.$$

În cazul general

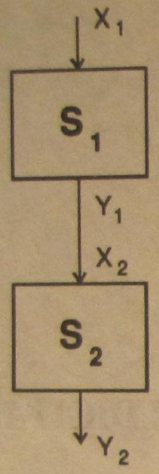



Figura 2.1

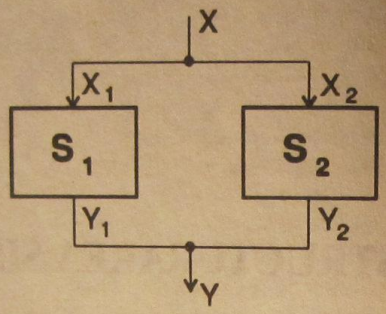



Figura 2.2

$$S = (X_1, Y_n, f),$$
$$f: X_1 \to P^*(Y_n),$$
$$f = f_1 \circ f_2 \circ \ldots \circ f_n.$$

• **Extensia paralel** este exemplificată, în cazul cel mai simplu, prin sistemul $S$, reprezentat în figura 2.2 şi definit prin

$$S = (X_1 \times X_2, Y_1 \times Y_2, f) = (X, Y, f)$$

$$f: X_1 \times X_2 \to P^*(Y_1 \times Y_2).$$

Practic, în acest caz nu apare nici o interacţiune între cele două subsisteme, $S_1$ şi $S_2$. Evident, în cazul general

$$S = (X_1 \times X_2 \times \ldots \times X_n, Y_1 \times Y_2 \times \ldots \times Y_n, f) = (X, Y, f),$$
$$f: X_1 \times X_2 \times \ldots \times X_n \to P^*(Y_1 \times Y_2 \times \ldots \times Y_n).$$

• **Extensia serie - paralel** reprezintă o combinaţie a celor două anterior prezentate. Varianta cea mai simplă este reprezentată în figura 2.3.

• **Introducerea unei bucle de reacţie** este o conexiune care determină noi comportamente într-un sistem digital. Configuraţia cea mai simplă pe care o imaginăm este cea din figura 2.4 în care pornind de la sistemul

$$S = (X \times X', Y, f')$$

cu

$$f': X \times X' \to P^*(Y),$$



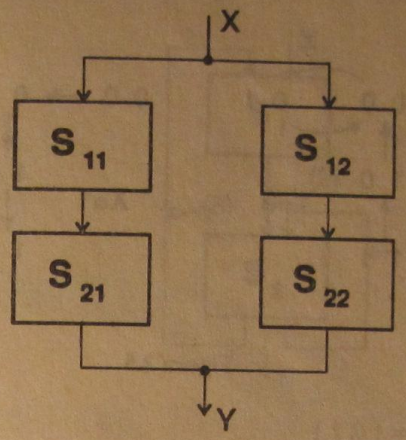



Figura 2.3

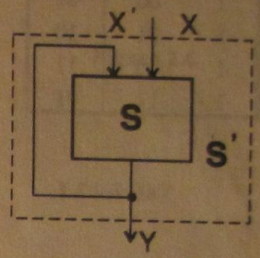



Figura 2.4

se poate obţine sistemul

$$S' = (X, Y, g)$$
$$g: X \to P^*(Y),$$

conectînd ieşirea la intrarea $X'$ ($X' = Y$). Sistemul $S$ poate fi unul rezultat prin mecanismele de structurare anterior expuse.

Există o diferenţă între funcţiile $f'$ şi $g$? Diferenţa este dată de comportamentul sistemului cu "intrarea constantă", în sensul că la intrare configuraţia binară nu-şi modifică valoarea.

Pentru $(x_i, x'_j) \in X \times X'$ aplicat constant la intrare vom regăsi, prin funcţia $f'$, la ieşire elemente din submulţimea $Y_k \subset Y$, deci *un singur* element din $P^*(Y)$. Dar pentru $x_i \in X$ vor rezulta la ieşirea sistemului $S'$ submulţimile $P^*_i(Y) \subset P^*(Y)$, adică *mai multe* elemente din $P^*(Y)$. Într-adevăr, explicitînd pe $f'$ rezultă

$$f': X \times Y \to P^*(Y)$$

iar pentru $x_i \in X$ identic pe intrare avem

$$f'_i: \{x_i\} \times Y \to P^*(Y)$$

ceea ce poate fi echivalat prin

$$f'_i: Y \to P^*(Y).$$

Ultima formă sugerează *comportamentul autonom* al sistemului în contextul lui $x_i$. *Sistemul are un comportament pe ieşire fără ca acesta să fie o consecinţă a unei modificări a intrării.*

**Exemplul 2.1** Vom particulariza sistemul $S$ după cum urmează:

| X × X' | Y |
|---|---|
| 0 00 | 01 |
| 0 01 | 00 |
| 0 10 | 00 |
| 0 11 | 10 |
| 1 00 | 01 |
| 1 01 | 10 |
| 1 10 | 11 |
| 1 11 | 01 |

**Tabelul 2.1**

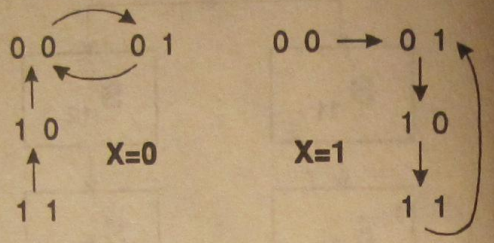



**Figura 2.5**

$$X = \{ 0, 1 \}$$
$$X' = \{ 00, 01, 10, 11 \}$$
$$Y = \{ 00, 01, 10, 11 \}$$

iar funcţia de transfer $f'$ definită prin tabelul 2.1.

Prin $f'$ fiecărei configuraţii binare de intrare îi corespunde un singur element din $Y$. Spre exemplu dacă intrarea este *010* la ieşire rezultă univoc *00*. Dar, prin funcţia $g$, unei configuraţii de intrare îi poate corespunde orice element din $Y$, astfel încît

$$g(0) = Y,$$
$$g(1) = Y.$$

Cu alte cuvinte, aplicînd *1* pe intrarea sistemului $S'$ vom putea găsi la ieşire, la momente de timp diferite, orice element din $Y$. Grafurile după care evoluează ieşirea sistemului $S'$ sub intrare constantă sînt reprezentate în figura 2.5. Pentru $X = 0$, după eventuale două tranziţii (din *11* în *10* şi din *10* în *00*) ieşirea sistemului va evolua într-un ciclu ce cuprinde configuraţiile binare *00* şi *01*. Pentru $X = 1$ ciclul cuprinde trei valori din $Y$ după o eventuală tranziţie din *00* în *01*.

Deci cu intrări constante, sub acţiunea buclei de reacţie numai, ieşirea sistemului evoluează în virtutea autonomiei astfel dobîndite. □

*În concluzie* putem afirma că extensiile serie, paralel şi serie-paralel nu introduc comportamente suplimentare, drept urmare a conexiunilor prin care se configurează, iar conectarea în buclă determină creşterea autonomiei sistemului faţă de autonomia anterioară introducerii buclei. În exemplul dat, autonomia sistemului $S$ era nulă, dar în cazul general, al unor sisteme mai complexe, ea poate exista şi înaintea introducerii buclei peste $S$, drept urmare a unor bucle interne preexistente. Efectul evidenţiat sugerează un principiu de ordonare sau clasificare a sistemelor digitale.

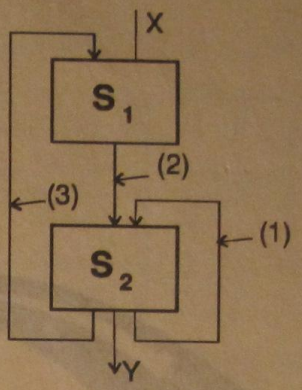



**Figura 2.6**

● **Clasificarea sistemelor digitale.**

**Definiţia 2.1** Două bucle de reacţie se *includ* atunci cînd peste un sistem extins serie, format din cel puţin unul care are o buclă, se mai închide o buclă, despre care spunem că o include pe prima. □

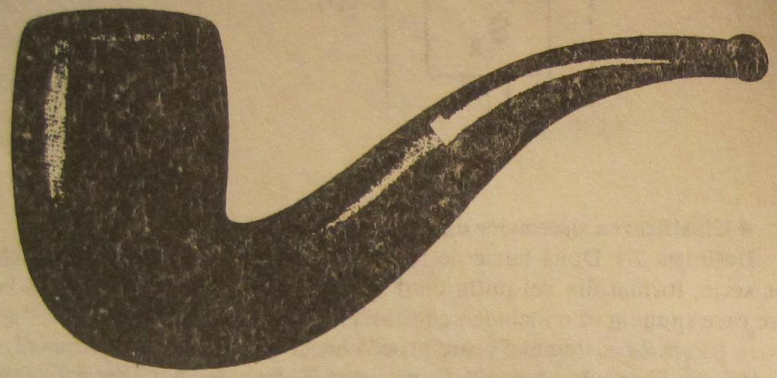
În figura 2.6 sistemul $S_2$ care posedă bucla închisă prin conexiunea *(1)* este extins serie cu $S_1$ prin legătura *(2)*. Conexiunea *(3)* închide o nouă buclă despre care spunem că o include pe cea dată de *(1)*.

**Definiţia 2.2** Un sistem este de *ordinul N* (notăm cu *SON*) dacă în el pot fi puse în evidenţă $N$ bucle ierarhizate prin incluziune. □

**Definiţia 2.3** Circuitele logice combinaţionale sînt sisteme de ordinul zero. □

Atunci cînd cardinalul mulţimilor din $P^*(Y)$ este mai mare decît unu înseamnă că, aplicînd o valoare din $X$ pe intrare putem obţine la ieşire, la momente de timp diferite, valori diferite din $Y$. Acest fapt corespunde autonomiei sistemului, autonomie dobîndită prin efectul buclelor interne. Dacă sistemul este definit astfel încît cardinalul mulţimilor din $P^*(Y)$ este unu, atunci oricărei valori aplicate la intrare îi corespunde numai una singură la ieşire. Sîntem deci în cazul în care $f_c : X \to Y$, ce corespunde circuitelor combinaţionale, pentru care ieşirea este univoc determinată de intrare.

Teoretic putem vorbi de o ierarhizare pe un număr nelimitat de niveluri, dar practic, procesul de structurare va introduce efecte care vor limita demersul nostru, în contextul tehnologic şi conceptual actual, în zona *SO4 - SO5*.



*Ceci n'est pas une pipe.*

R. Magritte : *Ceci n'est pas une pipe.*

3

# CIRCUITE LOGICE COMBINAŢIONALE
## Sisteme de ordinul zero

## 3.1 Descrierea formală a circuitelor logice combinaţionale

### 3.1.1 Variabile şi funcţii logice

Conform definiţiilor 1.3 şi 1.4 în figura 3.1 este reprezentat un circuit logic combinaţional *(CLC)* cu $n$ intrări şi $m$ ieşiri. $X$ reprezintă *variabila de intrare* care poate lua valorile $X_1, X_2, ..., X_p$ cu $p = 2^n$ iar $Y$ este *variabila de ieşire* cu valorile posibile $Y_1, Y_2, ..., Y_q$ pentru $q = 2^m$. Prin funcţia $f$ oricărui $X_i$ îi corespunde un $Y_j$. Aplicînd o valoare pe intrare, într-un interval de timp nul, pentru acest model teoretic, şi cît mai mic în cazul unor circuite reale, se obţine la ieşire o valoare pentru $Y$.

Numim simbolurile $x_i$ *variabilă binară de intrare* şi $y_j$ *variabilă binară de ieşire*. Aceste variabile iau valori în mulţimile $\{0, 1\}$ sau {*adevărat, fals*}.

Astfel, putem spune că $X$ ia valoarea *10010* şi $Y$ ia valoarea *101*, pentru un *CLC* la care $n = 5$ şi $m = 3$. Sau ne putem exprima în sensul că pe *intrarea* $x_3$ se aplică *0*, adică variabila de intrare $x_3$ ia valoarea *0*.

Ce *funcţii* poate avea asociate un *CLC* ? Începînd cu cele mai simple, aceste funcţii pot fi listate. Pentru exemplificare vom considera cazul *CLC*-urilor cu o *singură ieşire*, pentru care $m = 1$. O vom nota cu $y$.

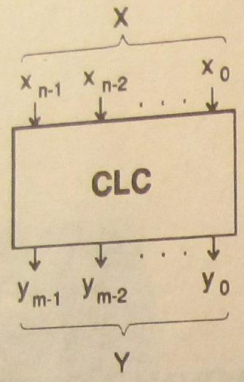



**Figura 3.1**

| x | $f_0^1$ | $f_1^1$ | $f_2^1$ | $f_3^1$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 |

**Tabelul 3.1**

• **Circuitele logice cu o intrare** pot avea, conform tabelului 3.1, patru funcţii de transfer. Vom nota unica intrare cu $x$.

Funcţiile $f_0^1$ şi $f_3^1$ iau aceeaşi valoare indiferent de valoarea aplicată intrării. Cea de a doua se numeşte *tautologie* deoarece ia valoarea logică *adevărat* tot timpul, indiferent de intrare. Funcţia $f_1^1$ repetă la ieşire valoarea intrării. Singura care asigură o "prelucrare" a intrării pare să fie $f_2^1$ care substituie o valoare binară cu cealaltă, transformă *adevăratul* în *fals* şi invers sau pe *0* în *1* şi invers. *Negaţia adevăratului* este *falsul* iar *falsul* negat generează *adevăratul*. Vom numi această funcţie *negaţie* sau *NOT* şi vom nota $f_2^1 = x'$. Altfel, vom spune că $y = x'$ dacă *CLC*-ul are funcţia *NOT*.

Recapitulînd vom scrie:

$$f_0^1(x) = 0,$$
$$f_1^1(x) = x,$$
$$f_2^1(x) = x',$$
$$f_3^1(x) = 1.$$

• **Circuitele logice cu două intrări** (şi o ieşire) sînt definite în tabelul 3.2. Ele formează clasa funcţiilor logice binare. Variabila de intrare, $X$ are în acest caz patru valori şi este specificată prin două variabile binare $x_1$ şi $x_0$. Cum $y$ trebuie să ia valori binare pentru cele patru valori aplicate intrării, vor rezulta *16* comportamente distincte ale ieşirii, deci $f_0^2, f_1^2, ..., f_{15}^2$.

Remarcăm în mulţimea funcţiilor cu două intrări cîteva mai deosebite. Astfel $f_1^2$ ia valoarea *1* numai dacă $x_1$ şi $x_0$ iau valoarea *1*. Funcţia $f_6^2$ ia valoarea *1* numai dacă $x_1$ şi $x_0$ sînt diferite. Funcţia $f_9^2$ este inversul (negaţia) funcţiei $f_6^2$ luînd valoarea *1* pentru cazul în care variabilele $x_1$ şi $x_0$ iau valori identice. Mai remarcăm



funcţia $f_7^2$ care ia valoarea *1* dacă cel puţin una din variabilele de intrare este egală cu *1*.

Regăsim şi aici funcţia *NOT* prin $f_{10}^2 = x_0'$ şi $f_{12}^2 = x_1'$.

| $x_1$ $x_0$ | $f_0^2$ | $f_1^2$ | $f_2^2$ | $f_3^2$ | $f_4^2$ | $f_5^2$ | $f_6^2$ | $f_7^2$ | $f_8^2$ | $f_9^2$ | $f_{10}^2$ | $f_{11}^2$ | $f_{12}^2$ | $f_{13}^2$ | $f_{14}^2$ | $f_{15}^2$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 0 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 1 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 1 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |

**Tabelul 3.2**

• **Circuitele logice cu trei intrări** pot fi şi ele similar întabelate dar ar rezulta un tabel cu *8* rînduri şi *256* de coloane în partea din dreapta. Într-adevăr cu trei variabile binare de intrare, $x_2, x_1, x_0$, se pot construi *8* valori de intrare iar numărul de funcţii distincte rezultat este de $2^8 = 256$. Se poate uşor deduce că numărul de funcţii distincte cu $n$ variabile binare de intrare este

$$N = 2^{2^n}.$$

Această dependenţă exponenţială nu ne va permite tratarea distinctă a funcţiilor logice, impunînd procedee de abordare care pornind de la *funcţiile logice binare* (sau de la o parte a acestora) să se poată exprima toate celelalte funcţii. Se impune deci un aparat formal bazat pe axiome simple şi puţine în virtutea cărora să poată fi descrisă, analizată şi stăpînită imensa diversitate a circuitelor logice ce se întrevede în acest paragraf. *Listării* funcţiilor îi vom substitui *mecanismul de generare* al aceloraşi funcţii.

## 3.1.2 Conectivi logici

Se pune întrebarea: putem găsi un număr de funcţii cu ajutorul cărora să putem exprima *toate* celelalte funcţii logice? Se pare că da. În mulţimea funcţiilor logice binare am evidenţiat deja unele pe care vom încerca să le folosim în acest scop.

• **Funcţia $f_1^2$, numită *ŞI* sau *AND*,** poate fi exprimată folosind *conectivul logic* $\cdot$ sau $\wedge$, astfel

$$f_1^2(x_1, x_0) = x_1 \wedge x_0 = x_1 \cdot x_0$$

sau pur şi simplu

$$f_1^2(x_1, x_0) = x_1 x_0;$$

această ultimă formă fiind cea pe care o vom utiliza în continuare. Denumirea este evidentă, deoarece $f_1^2$ ia valoarea *1* numai dacă $x_1$ şi $x_0$ iau această valoare. O altă

interpretare porneşte de la faptul că o propoziţie, obţinută prin compunerea a două propoziţii utilizînd conectivul şi, este *adevărată* dacă şi numai dacă ambele propoziţii sînt adevărate. Spre exemplu, propoziţia

*"Ion este înalt* şi *Mihai este gras"*

este adevărată numai dacă propoziţiile

*"Ion este înalt"*

şi

*"Mihai este gras"*

sînt amîndouă adevărate.

• Funcţia $f_7^2$ poate fi numită *SAU* sau *OR* şi exprimată cu ajutorul conectivului logic + sau ∨, după cum urmează:

$$f_7^2(x_1, x_0) = x_1 \vee x_0 = x_1 + x_0,$$

ultima formă fiind cea curent folosită în continuare.

Justificarea denumirii este dată de faptul că $f_7^2$ ia valoarea *1* dacă $x_1$ sau $x_0$ are această valoare. Folosind exemplul cu propoziţii, anterior dat, vom spune că propoziţia

*"Ion este înalt* sau *Mihai este gras"*

este adevărată dacă cel puţin una din propoziţiile ce o compun este adevărată.

• Dacă la funcţiile *AND* şi *OR* adăugăm funcţia *NOT* vom reuşi să exprimăm cu ajutorul lor *orice* funcţie logică. Demonstraţia este imediată şi se poate baza pe exemplul oferit de funcţia logică definită în tabelul 3.3. În ce condiţii $y_1$ ia valoarea *1*? Ieşirea $y$ este egală cu *1* dacă "$x_2$ este zero şi $x_1$ este unu şi $x_0$ este zero sau $x_2$ este unu şi $x_1$ este zero şi $x_0$ este zero". *Ţinem cont de faptul că* "$x$ este zero" este echivalent cu faptul că "$x'$ este unu". În aceste condiţii putem scrie:

$$y_1 = x_2' x_1 x_0' + x_2 x_1' x_0'.$$

| $x_2$ | $x_1$ | $x_0$ | $y_1$ | $y_0$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 | 0 |

**Tabelul 3.3**



Observăm faptul că *orice funcţie* poate fi scrisă ca o "sumă" (*OR*) de "produse" (*AND*) formate din variabile ($x_i$) sau variabile negate ($x_i'$).

Un exerciţiu în plus îl facem cu funcţia $y_0$ pentru care scriem:

$$y_0 = x_2' x_1' x_0' + x_2' x_1' x_0 + x_2 x_1' x_0 + x_2 x_1 x_0'.$$

• Un alt conectiv logic poate fi introdus pentru funcţia $f_6^2$, care este un fel de *OR* ce *exclude* cazul în care ambele variabile de intrare sînt adevărate (sau iau valoarea unu). Este vorba de funcţia *SAU-EXCLUSIV* sau *XOR* pe care o extragem din tabelul 3.2 sub forma

$$f_6^2(x_1, x_0) = x_1 x_0' + x_1' x_0 = x_1 \oplus x_0.$$

Am notat prin ⊕ conectivul *XOR*.

Această funcţie comportă mai multe interpretări.

Una ar fi cea de *sumă modulo doi*. Definiţia prin tabel confirmă acest lucru deoarece *0 plus 0 = 0, 0 plus 1 = 1 plus 0 = 1* iar *1 plus 1 = 0*, în cazul cînd sumăm modulo doi, în baza de numeraţie doi.

O altă interpretare este aceea de circuit de inversare (complementare) comandat. Observăm că

$$x_1 \oplus x_0 \mid_{x_1=0} = x_0,$$

şi

$$x_1 \oplus x_0 \mid_{x_1=1} = x_0';$$

deci variabila $x_1$ poate comanda tranziţia prin circuit a valorii lui $x_0$ neschimbată (pentru $x_1 = 0$) sau a valorii lui $x_0$ complementată (pentru $x_1 = 1$).

Ultima interpretare ar fi cea de *funcţie de anticoincidenţă* deoarece *XOR* de $x_1$ şi $x_0$ ia valoarea *1* atunci cînd valoarea celor două variabile nu coincide.

Toate aceste interpretări sînt utile pentru a sugera aplicaţii ale acestei funcţii. Sumarea, complementarea sau comparaţia pot fi realizate pornind de la această funcţie elementară. Descrierea formală este bine să fie, de cîte ori este posibil, completată de interpretări ce ne duc cu gîndul la aplicaţii posibile.

• Putem combina simplu conectivii evidenţiaţi pentru a defini funcţiile *NAND* (şi-nu), *NOR* (sau-nu) şi *NXOR* (coincidenţă).

Scriem în consecinţă:

$$f_{14}^2(x_1, x_0) = (x_1 x_0)' \quad \text{pentru } NAND,$$
$$f_8^2(x_1, x_0) = (x_1 + x_0)' \quad \text{pentru } NOR,$$
$$f_9^2(x_1, x_0) = (x_1 \oplus x_0)' \quad \text{pentru } NXOR.$$

Se observă pornind de la tabelul 3.2, că

$$x_1 \oplus x_0' = x_1' \oplus x_0 = (x_1 \oplus x_0)'$$

de unde rezultă şi

$$x_1' \oplus x_0' = x_1 \oplus x_0$$

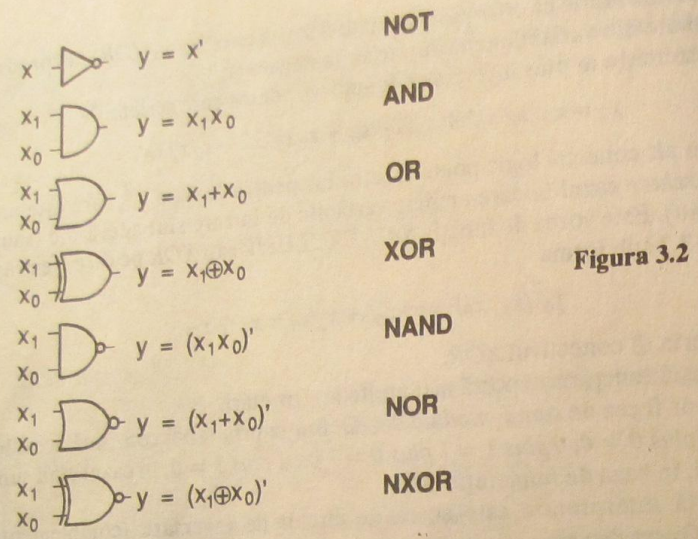



**Figura 3.2**

Vom putea studia mai uşor proprietăţile funcţiilor logice in contextul unei teorii formale a algebrei logice.

• Reprezentarea circuitelor asociate conectivilor logici anteriori descrişi este dată in figura 3.2.

### 3.1.3 Formalizarea algebrei logice

Logica matematică, sau algebra logică, a fost întemeiată prin lucrările lui George Boole la mijlocul secolului trecut. În aceeaşi perioadă contribuţii importante a avut şi De Morgan. Un rol decisiv în aplicarea acestei teorii la studiul ingineresc al sistemelor digitale l-a avut Claude Shannon. Contribuţii remarcabile în această direcţie a avut şi şcoala românească de matematică prin acad. Grigore Moisil şi colaboratorii săi.

Vom trece în revistă pe scurt cele patru probleme principale pe care încearcă să le rezolve o teorie a funcţiilor logice.

• **Problema posibilităţii deductive** are în vedere premizele şi concluziile unei expresii logice. Este importantă facilitatea care să ne permită transformarea unei expresii logice în alte expresii logice echivalente. Realizarea concretă a unui circuit asociat unei expresii va fi eficientă numai pentru anumite forme. De asemenea, inginerul trebuie să poată impune diverse criterii în implementare, criterii care sînt indeplinite mai bine numai de anumite forme pe care le pot lua expresiile logice.

• **Problema echivalenţei** se referă la posibilitatea de a decide dacă două funcţii logice sînt echivalente, in sensul că, pentru aceleaşi configuraţii de intrare iau aceleaşi valori pe ieşire.

• **Problema minimizării** se bazează pe rezolvarea celor două anterioare. Fiind dată o funcţie logică trebuie găsită o formă *echivalentă*, prin *deducţie*, astfel



încît să conţină un număr minim de simboluri. Unei expresii logice minimizate îi corespunde, *de regulă*, o implementare minimală; dar numai de regulă, pentru că există situaţii în care datorită unor restricţii tehnologice minimul algebric nu corespunde cu cel fizic.

• **Problema deciziei** se referă la decizia asupra caracterului *tautologic* al unei funcţii logice sau asupra *realizabilităţii* acesteia. O tautologie este o funcţie care ia valoarea logică adevărat pentru *orice* configuraţie binară de intrare. O funcţie logică este realizabilă dacă ia valoarea logică de adevărat pentru *cel puţin* o configuraţie binară de intrare.

• **Teoreme fundamentale**. Dezvoltarea axiomatică a algebrei logice prezintă, pentru noi un interes pur teoretic. Există teorii care pornesc de la o singură axiomă. Există teorii dezvoltate cu un singur conectiv logic (*NAND* sau *NOR*). Pentru acest curs sînt importante numai teoremele esenţiale ce s-au impus. Ele vor fi listate în continuare. Demonstrarea lor este evidentă. Toate teoremele vor fi enunţate sub forma unor *echivalenţe*. Pentru echivalenţă vom utiliza simbolul =, ca în exemplul ce urmează

$$f(A, B) = g(B).$$

Evităm o tratare axiomatică nesemnificativă pentru o abordare inginerească.

**Propoziţia 3.1** Conform principiului identităţii putem scrie

$$A = A. \square$$

Acest principiu fundamental se află împreună cu altele la originea limitelor funcţionale în domeniul sistemelor digitale. Rigiditatea impusă de acesta goleşte de "viaţă" demersul inginerului oricît de "inteligente" ar deveni maşinile sale. În realitate un astfel de principiu este absurd dacă este dus pînă la consecinţe ultime. În lumea sistemelor numerice este acceptat pentru ca acestea să poată fi realizate corect, să poată fi multiplicate şi să poată fi folosite într-o manieră riguros previzibilă. Limitele maşinilor astfel concepute şi realizate sînt date de limitele în care acest principiu nu-şi manifestă absurditatea.

**Propoziţia 3.2** Principiul dublei negaţii este dat de următoarea echivalenţă

$$(A')' = A. \square$$

Dubla negare a unei variabile nu afectează valoarea acesteia. Şi acest principiu este limitativ. Există logici care nu-l acceptă (logica intuiţionistă). În realitate acest principiu poate fi consistent numai în situaţii limită. Maşinile digitale îl presupun, limitîndu-se astfel fundamental.

**Propoziţia 3.3** Legea sau teorema *asociativităţii* este dată de următoarele expresii

$$x_1+(x_2+x_3) = (x_1+x_2)+x_3 = x_1+x_2+x_3,$$

$$x_1 (x_2 x_3) = (x_1 x_2) x_3 = x_1 x_2 x_3,$$

$$x_1 \oplus (x_2 \oplus x_3) = (x_1 \oplus x_2) \oplus x_3 = x_1 \oplus x_2 \oplus x_3. \square$$

**Propoziţia 3.4** *Comutativitatea* este definită prin

$$x_1+x_2 = x_2+x_1,$$
$$x_1 x_2 = x_2 x_1,$$
$$x_1 \oplus x_2 = x_2 \oplus x_1. \square$$

**Propoziţia 3.5** *Distributivitatea* este definită prin

$$x_1 (x_2+x_3) = (x_1 x_2)+(x_1 x_3) = x_1 x_2+x_1 x_3,$$
$$x_1(x_2 \oplus x_3) = (x_1 x_2) \oplus (x_1 x_3) = x_1 x_2 \oplus x_1 x_3,$$
$$x_1+x_2 x_3 = (x_1+x_2)(x_1+x_3). \square$$

**Propoziţia 3.6** Legile lui *De Morgan* sînt

$$(x_1 + x_2)' = x_1' x_2',$$
$$(x_1 x_2)' = x_1' + x_2'. \square$$

**Propoziţia 3.7** Regulile de *absorbţie* sînt

$$x_1 + x_1' = 1. \text{ (terţul exclus)},$$
$$x_1 + x_1 = x_1,$$
$$x_1 x_1' = 0,$$
$$x_1 x_1 = x_1,$$
$$x_1 + x_1 x_2 = x_1,$$
$$x_1(x_1 + x_2) = x_1. \square$$

**Propoziţia 3.8** Regulile de *semiabsorbţie* sînt

$$x_1 + x_1' x_2 = x_1 + x_2,$$
$$x_1(x_1' + x_2) = x_1 x_2. \square$$

**Propoziţia 3.9** Legile *excluderii* sînt

$$x_1 x_2 + x_1' x_2 = x_2,$$
$$(x_1+x_2)(x_1+x_2') = x_1. \square$$

**Propoziţia 3.10** Prin substituirea cu valori obţinem

$$x_1 + 0 = x_1,$$
$$x_1 + 1 = 1,$$
$$x_1 0 = 0,$$
$$x_1 1 = x_1,$$
$$x_1 \oplus 0 = x_1,$$
$$x_1 \oplus 1 = x_1'. \square$$



**Exemplul 3.1** Se dă funcţia

$$f(x_1, x_2, x_3) = x_1' x_2 x_3 + x_1' x_2' x_3 + x_1' x_2' x_3'$$

şi se cere să se aducă la o formă minimă.

Aplicînd legea distributivităţii obţinem

$$f(x_1, x_2, x_3) = x_1' x_3 (x_2+x_2') + x_1' x_2' x_3';$$

prin absorbţie (terţul exclus) rezultă

$$f(x_1, x_2, x_3) = x_1' x_3 + x_1' x_2' x_3',$$

şi aplicînd din nou distributivitatea

$$f(x_1, x_2, x_3) = x_1' (x_3+x_2' x_3').$$

Conform regulilor de semiabsobţie se obţine

$$f(x_1, x_2, x_3) = x_1' (x_3+x_2')$$

care este forma minimizată. □

- **Formele canonice de reprezentare** ale funcţiilor logice sînt

- forma normală disjunctivă (*FND*),

- forma normală conjunctivă (*FNC*).

*FND* este obţinută ca o *sumă de produse* (*SAU* de *ŞI*-uri) în care fiecare produs (*mintermen*) depinde de *toate* variabilele binare de intrare.

*FNC* este obţinută ca un *produs de sume* (*SI* de *SAU*-uri) în care fiecare sumă (*maxtermen*) depinde de *toate* variabilele binare de intrare.

**Exemplul 3.2** Se consideră funcţia

$$g = x_1 \oplus x_2 \oplus x_3.$$

*FND* a funcţiei *g* este

$$g = x_1' x_2' x_3 + x_1' x_2 x_3' + x_1 x_2' x_3' + x_1 x_2 x_3$$

iar *FNC* este

$$g = (x_1 + x_2 + x_3)(x_1 + x_2' + x_3')(x_1' + x_2 + x_3')(x_1' + x_2' + x_3).$$

Un exemplu de mintermen din *g* este $x_1' x_2 x_3'$ şi unul de maxtermen este $(x_1 + x_2' + x_3')$. □

**Exemplul 3.3** Pornind de la o formă disjunctivă se poate construi forma *normală* disjunctivă, conform exemplului următor.

$$f(x_1, x_2, x_3) = x_1 x_2 + x_1 x_2' x_3 =$$
$$= x_1 x_2 (x_3 + x_3') + x_1 x_2' x_3 =$$
$$= x_1 x_2 x_3 + x_1 x_2 x_3' + x_1 x_2' x_3. \square$$

*FNC* poate fi obţinută, pornind de la o formă conjunctivă, într-o manieră similară.

*FND* şi *FNC* sînt formele cele mai complexe pe care le poate lua o funcţie logică. Ele sînt totuşi utile în cazul unor anumite tehnici de implementare.

**Propoziţia 3.11** Funcţia

$$f(x_1, x_2, ..., x_n) = x_1 \oplus x_2 \oplus ... \oplus x_n$$

are asociată o *FND* ce conţine $2^{n-1}$ mintermeni. □

*Demonstraţia* o propunem spre realizare cititorului.

### 3.1.4 Minimizarea funcţiilor logice

Forma cea mai complexă a unei funcţii logice este *FND* sau *FNC*. În continuare vom lucra, de regulă, cu *FND* deoarece aceasta este cea care se obţine cel mai uşor pornind de la reprezentările uzuale ale funcţiilor logice. (În paragraful 3.1.2 am exemplificat acest lucru pentru funcţiile definite în tabelul 3.3.)

• **Procedeele algebrice de minimizare** presupun folosirea, în principal, a legilor excluderii şi semiabsorbţiei.

Legea excluderii înlătură la fiecare aplicare un mintermen şi micşorează unul din produsele rămase cu o variabilă binară. Legea semiabsorbţiei permite micşorarea unui produs elementar cu o variabilă. Un prim exemplu a fost deja dat (Exemplul 3.1), în continuare vom da altele.

**Exemplul 3.4** Într-o primă etapă în expresia

$$h = x_1x_2x_3x_4 + x_1'x_2x_3x_4 + x_1x_2'x_3x_4 + x_1x_2x_3'x_4$$

se "taie" al doilea produs şi variabila $x_1$ din primul produs, drept urmare a unei aplicări a legii excluderii; iar în a doua etapă în expresia rămasă se taie $x_2'$ din cel de al doilea produs

$$h = x_2x_3x_4 + x_1x_2'x_3x_4 + x_1x_2x_3'x_4$$

conform legii semiabsorbţiei. În final al doilea produs micşorat permite o nouă aplicare a aceleaşi legi rezultînd forma minimizată

$$h = x_2x_3x_4 + x_1x_3x_4 + x_1x_2x_4 . \square$$

**Exemplul 3.5**

$$x_1x_2'x_3x_4' + x_1x_2x_3x_4' + x_1x_2'x_3x_4' + x_1x_2x_3x_4' + x_1x_2x_3x_4 =$$

(1), (3), (2), (4)

$$= x_1x_4' + x_1x_2x_3 .$$

Au fost numerotate tăieturile în ordinea executării lor. □

• **Diagramele Veitch - Karnaugh** permit o reprezentare care asigură o minimizare mai rapidă şi mai sigură.



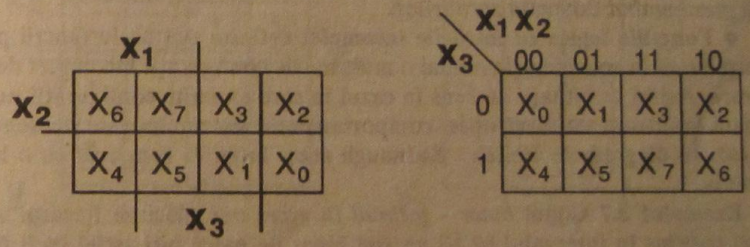
**Definiţia 3.1** Doi mintermeni spunem că sînt *adiacenţi* atunci cînd iau valoarea adevărat pentru configuraţii binare ce diferă prin valoarea unei singure variabile binare. □

**Exemplul 3.6** Următoarele perechi de mintermeni sînt adiacente:

$$x_1'x_2x_3'x_4, x_1'x_2x_3'x_4',$$
$$x_1x_2x_3x_4', x_1x_2x_3x_4,$$
$$x_1x_2'x_3', x_1'x_2'x_3' . \square$$

Existenţa termenilor adiacenţi într-o expresie permite aplicarea legii excluderii. Deci, construirea unei reprezentări care pune în evidenţă adiacenţele va permite găsirea unui mecanism care să asigure minimizarea funcţiilor logice. În figura 3.3 sînt reprezentate diagramele Veitch *(a)* şi Karnaugh *(b)* ce au cîte o locaţie pentru fiecare mintermen ce poate defini o funcţie logică de trei variabile. Locaţiile alăturate corespund mintermenilor adiacenţi. În fiecare locaţie am scris valoarea variabilei de intrare pentru care mintermenul ia valoarea adevărat. Pentru două şi patru variabile diagrama Veitch este reprezentată în figura 3.4

În diagrama Veitch din figura 3.4 am notat cu $x_1$ jumătatea din suprafaţă formată din primele două coloane în care sînt înscrişi mintermenii ce iau valoarea adevărat pentru $x_1 = 1$, cealaltă jumătate corespunzînd mintermenilor ce iau această valoare pentru $x_1 = 0$. Variabila binară $x_2$ împarte, pe acelaşi criteriu, suprafaţa în două părţi egale, cele două rînduri superioare $(x_2 = 1)$ şi cele două




a) b)

**Figura 3.3**

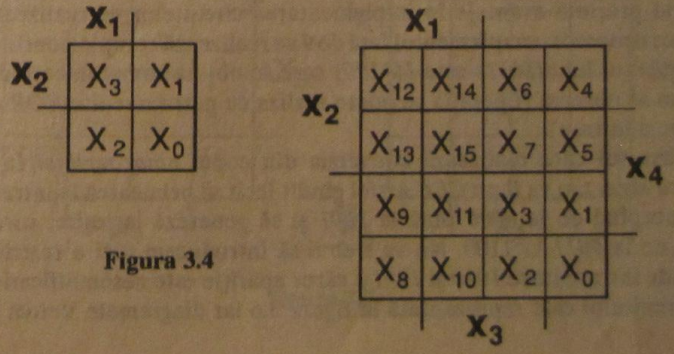



**Figura 3.4**

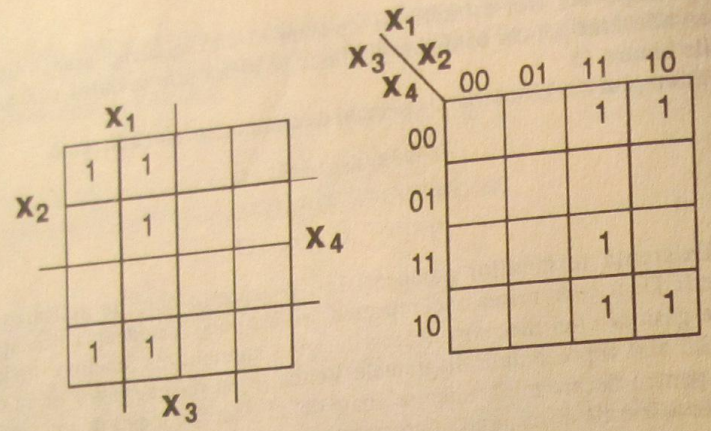



Figura 3.5

inferioare ($x_2 = 0$). La fel acţionează şi celelalte variabile divizînd suprafeţe în alte două moduri distincte. Similar poate fi explicată şi construirea diagramei Karnaugh.

Reprezentarea unei funcţii printr-o diagramă Veitch sau Karnaugh o vom exemplifica cu funcţia din Exemplul 3.5 în reprezentarea din figura 3.5 unde în locaţiile corespunzătoare mintermenilor prezenţi în expresie am scris *1*, în celelalte locaţii *presupunînd* existenţa zerourilor.

• **Funcţiile logice cu intrările incomplet definite** sînt acele funcţii pentru care este luată în consideraţie numai o mulţime de configuraţii incomplet definită la intrare. Astfel de situaţii au sens în cazul în care anumite configuraţii nu apar niciodată la intrare sau dacă apar, comportamentul sistemului este neesenţial în acest caz. În diagramele Veitch - Karnaugh acest lucru va fi marcat cu o liniuţă orizontală.

**Exemplul 3.7** Codul *binar - zecimal în exces trei* asociază fiecărui număr zecimal cuprins în intervalul [*0, 9*] un cod binar de patru biţi astfel încît fiecare număr zecimal este asociat cu unul binar la care se adaugă numărul trei. Astfel, pentru *0* (zero), în acest cod avem reprezentarea *0000 + 0011 = 0011*, pentru *1* avem *0100*, pentru *2* rezultă *0101* şi aşa mai departe pînă la *9* codificat prin *1100*. Acest cod prezintă avantaje în implementarea circuitelor ce realizează operaţii aritmetice deoarece complementul faţă de *9* se realizează complementînd bit cu bit. Complementul lui *8(1011)* este *1(0100)* care se obţine complementînd bit cu bit, după cum se observă (operaţia se poate realiza cu patru circuite *XOR* comandate comun pe o intrare).

Circuitul care realizează conversia din codul *binar-zecimal* în cel *binar-zecimal cu exces trei* va fi un *CLC* astfel gîndit încît să primească la intrare codurile binare începînd de la *0000* pînă la *1001* şi să genereze la ieşire, corespunzător codurile de la *0011* la 1100. Nu va trebui să introducem nici o restricţie pentru codurile de intrare între *1010* şi *1111* a căror apariţie este nesemnificativă. Schema bloc a circuitului este reprezentată în figura 3.6 iar diagramele Veitch asociate în



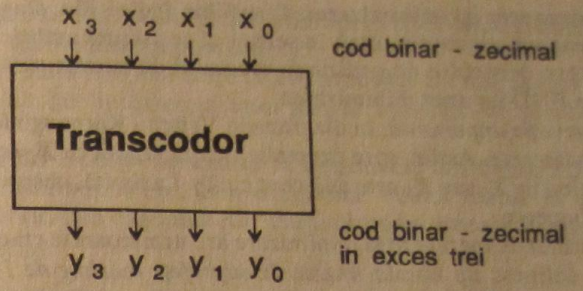



Figura 3.6

figura 3.7. În cele patru locaţii asociate valorii $X_i$ (figura 3.4) a fost înscris codul binar al numărului $i + 3$, pentru $i \in [0, 9]$ , în rest s-au înscris liniuţe ce pot lua *orice valoare binară*, pentru că valorile $X_{10} \ldots X_{15}$ nu vor apare niciodată la intrarea *Transcodor*-ului. □

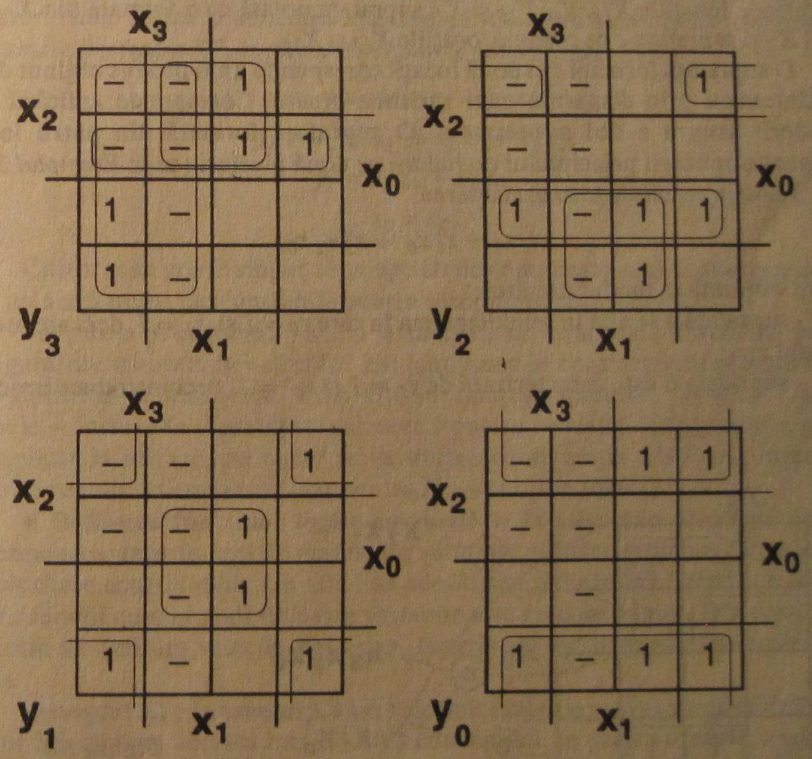



Figura 3.7

• **Extragerea şi minimizarea funcţiilor logice din diagramele Veitch - Karnaugh** este un proces simultan în sensul că se asigură ambele operaţii într-un singur pas spre deosebire de utilizarea tabelului de adevăr care presupune întîi extragerea în FND iar apoi minimizarea.

*O observaţie importantă*: în diagramele Veitch - Karnaugh *laturile opuse sînt considerate adiacente*. Astfel, spre exemplu locaţia notată cu $X_{13}$ este adiacentă cu cea care conţine pe $X_5$ sau $X_6$ este adiacent cu $X_2$. La limită, observăm că $X_0$, $X_8$, $X_4$ şi $X_{12}$ sînt adiacente.

Algoritmul de extragere şi minimizare are următoarele etape:

(1) se definesc un număr minim de *suprafeţe maxime de 1-uri* formate din *locaţii adiacente*, de *formă dreptunghiulară* avînd un *număr de locaţii care este o putere a lui 2*;

(2) se asociază fiecărei suprafeţe produsul elementar ce o caracterizează;

(3) se sumează aceste produse rezultînd funcţia logică.

(4) Diagramele asociate funcţiilor incomplet definite permit o minimizare avansată datorită faptului că în suprafeţele de *1*-uri pot fi incluse, *opţional*, şi locaţii marcate prin – , permiţînd prin aceasta maximizarea lor.

**Exemplul 3.8** În diagrama din figura 3.8 au fost puse în evidenţă trei suprafeţe *dreptunghiulare* formate din *2* şi *4* locaţii adiacente. Suprafaţa notată cu *a* ce cuprinde locaţiile $X_{14}$, $X_6$, $X_{15}$ şi $X_7$, suprafaţa notată cu *b* formată din $X_{14}$, $X_6$, $X_{10}$ şi $X_2$ şi suprafaţa *c* ce conţine locaţiile $X_{13}$ şi $X_{15}$.

O suprafaţă formată din două locaţii corespunde unui produs obţinut dintr-un mintermen prin dispariţia unei variabile binare. Corespunde aplicării legii excluderii asupra a doi mintermeni. O suprafaţă formată din patru locaţii corespunde aplicării principiului excluderii pe două niveluri (ca în *Exemplul 3.5*).

Funcţia rezultată este următoarea

$$x_1 x_2 + x_1 x_0' + x_3 x_2 x_0$$

şi a fost obţinută în modul următor:

- suprafaţa *a* se află în semidiagrama în care $x_2 = 1$ şi $x_1 = 1$, deci are asociat produsul $x_2x_1$;

- suprafaţa *b* este caracterizată de $x_1 = 1$ şi $x_0' = 1$, deci introduce produsul $x_1x_0'$;

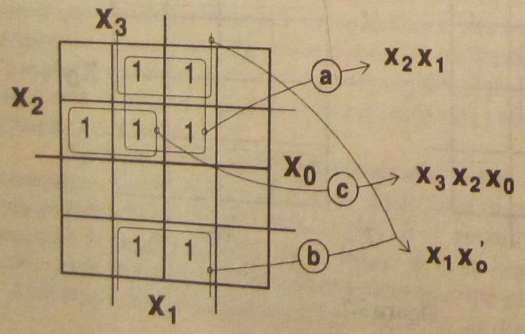



**Figura 3.8**



- suprafaţa *c* se află la intersecţia semisuprafeţelor caracterizate prin $x_3 = 1$, $x_2 = 1$ şi $x_0 = 1$, deci va adăuga produsul $x_3 x_2 x_0$ .

Cu cît suprafaţa este mai mare cu atît produsul are mai puţine componente pentru că rezultă din intersecţia unui număr mai mic de semisuprafeţe ale diagramei. □

**Exemplul 3.9** Funcţia introdusă în diagramele din figura 3.5 rezultă prin conturarea a două suprafeţe. Din diagrama Veitch extracţia este similară exemplului anterior. Din diagrama Karnaugh obţinem aceeaşi funcţie, şi anume

$$x_1 x_4' + x_1 x_2 x_3 ,$$

identică cu rezultatul minimizării algebrice din *Exemplul 3.5*. □

• **Extragerea şi minimizarea funcţiilor incomplet definite** se face respectînd şi punctul *(4)* din algoritmul de extragere. Folosirea opţională a locaţiilor cu conţinut indiferent permite maximizarea suprafeţelor, deci minimizarea numărului de elemente ale produselor.

Un mintermen cu valoare indiferentă inclus într-o suprafaţă capătă valoarea *1*, celelalte căpătînd, în consecinţă, valoarea *0*.

**Exemplul 3.10** Transcodorul definit în *Exemplul 3.7* poate fi realizat cu ajutorul unui circuit de sumare cu constanta *0011*. *Ţinînd cont de parţiala lui definire la intrare vom obţine o structură mai simplă. Conform suprafeţelor conturate în diagramele din figura 3.7, obţinem următoarele funcţii logice:*

$$y_3 = x_3 + x_2 x_1 + x_2 x_0,$$
$$y_2 = x_2 x_1' x_0' + x_2' x_0 + x_2' x_1,$$
$$y_1 = x_1 x_0 + x_1' x_0',$$
$$y_0 = x_0'.$$

Cititorul se poate singur convinge cît de complicate ar deveni aceste funcţii dacă nu am fi putut face opţiunile pentru locaţiile marcate cu – .

Se pune problema: putem afla cum se comportă circuitul pentru configuraţiile ignorate în definiţie? Evident, luînd în consideraţie opţiunea făcută prin definirea suprafeţelor. Vom citi suprafeţele acordînd valoarea *1* pentru semnele – incluse în suprafeţe şi valoarea *0* pentru celelalte. Spre exemplu, pentru $X_{11}$ aplicat la intrare, pe ieşire se va obţine configuraţia *1110* (din întîmplare conform regulii: în exces trei); iar pentru $X_{15}$ se obţine *1010*. □

• **Definirea funcţiilor logice cu variabile în tabel sau diagramă** permite reprezentarea unor funcţii de mai multe variabile utilizînd structuri formale de o complexitate controlabilă. Un tabel de adevăr sau o diagramă Veitch - Karnaugh pentru funcţii cu mai mult de patru variabile este greu de mînuit. Din acest motiv preferăm să definim valorile de ieşire funcţie de variabile sau chiar de funcţii logice.

**Exemplul 3.11** În tabelul 3.4 am redefinit funcţiile $y_1$ şi $y_0$ anterior definite în tabelul 3.3, de data aceasta într-o formă condensată, în partea dreaptă a tabelului introducînd, pe lîngă valori, şi funcţii de variabila $x_0$.

| $x_2$ | $x_1$ | $y_1$ | $y_0$ |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 1 | $x_0'$ | 0 |
| 1 | 0 | $x_0'$ | $x_0$ |
| 1 | 1 | 0 | $x_0'$ |

**Tabelul 3.4**

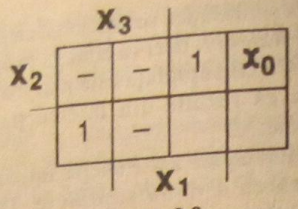



**Figura 3.9**

Compresia am făcut-o observînd că pentru $x_2 = x_1 = 0$, în tabelul 3.3, $y_1 = 0$ şi $y_0 = 1$; am completat în consecinţă în tabelul 3.4. Pentru $x_2 = 0$ şi $x_1 = 1$, în tabelul 3.3 se observă că $y_1 = x_0'$ iar $y_0 = 0$, ş.a.m.d. □

**Exemplul 3.12** Similar, putem comprima diagrama pentru $y_3$ din figura 3.7 sub forma diagramei din figura 3.9, unde fiecărei locaţii îi corespund două din prima reprezentare. Astfel pentru $X_0$ din ultima diagramă corespund $X_0$ şi $X_1$ din prima, ş.a.m.d. pînă la $X_7$ ce corespunde la $X_{14}$ şi $X_{15}$. În diagrama redusă, în locaţia $X_2$ am completat cu $x_0$ deoarece $X_5 = 1$ şi $X_4 = 0$ în organigrama de pornire iar $X_5$ se află în zona în care $x_0 = 1$ iar $X_4$ în zona în care $x_0 = 0$. Într-o situaţie inversă (cu $X_5 = 0$ şi $X_4 = 1$) am fi completat cu $x_0'$. □

• **Definiţia recursivă a funcţiilor logice** este sugerată de exemplele anterioare.

**Definiţia 3.2** Orice funcţie logică $f_i^j$ de $j$ variabile poate fi definită după cum urmează

$$f_i^j = f_k^{j-1} \cdot x_0 + f_l^{j-1} \cdot x_0'$$

unde

$$f_0^0 = 0 \text{ şi } f_1^0 = 1$$

fiind îndeplinite condiţiile ca

$$i \in [0, 2^{2^j}) \text{ iar } k, l \in [0, 2^{2^{j-1}}) . \square$$

Definiţia anterioară poate fi reluată şi în alte forme, forme ce sugerează definirea unei funcţii logice prin alte funcţii logice, astfel încît să poată fi utilizate reprezentări pentru funcţii cu mai multe variabile.

**Definiţia 3.3** Orice funcţie logică $f_i^j$ de $j$ variabile poate fi definită astfel

$$f_i^j = f_k^{j-2} x_1 x_0 + f_l^{j-2} x_1' x_0 + f_m^{j-2} x_1 x_0' + f_n^{j-2} x_1' x_0'$$

fiind îndeplinite condiţiile

$$i \in [0, 2^{2^j}) \text{ iar } k, l, m, n \in [0, 2^{2^{j-2}})$$

iar

$$f_0^0 = 0 \text{ şi } f_1^0 = 1. \square$$

Conform definiţiei anterioare putem folosi o reprezentare pentru funcţii de două variabile ($x_1$ şi $x_0$) pentru funcţii de $j$ variabile, dacă în tabel sau diagrame introducem funcţii de $j$-2 variabile. În cazul general putem da următoarea definiţie.



**Definiţia 3.4** Orice funcţie logică $f_i^j$ de $j$ variabile poate fi definită prin

$$f_i^j = f_k^{j-n} x_{n-1} x_{n-2} \ldots x_0 + \ldots + f_l^{j-n} x_{n-1}' x_{n-2}' \ldots x_0'$$

unde

$$f_0^0 = 0 \text{ şi } f_1^0 = 1$$

iar

$$i \in [0, 2^{2^j}) \quad \text{iar } k, \ldots, l \in [0, 2^{2^{j-n}}) . \square$$

*În virtutea acestei ultime definiţii se poate folosi pentru o funcţie de j variabile o reprezentare utilizată pentru n variabile.* Dacă $n=j$ sîntem în situaţia unei definiţii explicite prin care sînt indicate valori binare pentru configuraţiile de intrare. Astfel, dacă $j>n$, valorile ce definesc ieşirea sînt calculate prin funcţii de *alte* $j$-$n$ variabile decît $x_{n-1}, x_{n-2}, \ldots, x_0$. Pentru $n=2$ avem tabelul de adevăr din tabelul 3.5, din care se extrage relaţia dată în *Definiţia 3.3.*

| $x_1$ | $x_0$ | $f_i^j$ |
|---|---|---|
| 0 | 0 | $f_n(x_{j-1}, x_{j-2}, \ldots, x_2)$ |
| 0 | 1 | $f_m(x_{j-1}, x_{j-2}, \ldots, x_2)$ |
| 1 | 0 | $f_l(x_{j-1}, x_{j-2}, \ldots, x_2)$ |
| 1 | 1 | $f_k(x_{j-1}, x_{j-2}, \ldots, x_2)$ |

**Tabel ul 3.5**

• **Extragerea unei funcţii logice din diagrame Veitch-Karnaugh în care au fost introduse, pe lîngă valori, şi funcţii logice.**

Algoritmul de extragere comportă două etape :

(1) se extrage funcţia logică introducînd valoarea *0* în locaţiile în care se află funcţii logice;

(2) la funcţia astfel obţinută se sumează logic (*OR*) funcţia obţinută prin reintroducerea în tabel a funcţiilor şi considerarea *1*-rilor, din tabel ca fiind – ; suprafeţele se vor defini astfel încît să cuprindă aceeaşi funcţie iar valoarea lor se va înmulţi logic (*AND*) cu produsul asociat suprafeţei.

**Exemplul 3.13** Considerăm funcţia definită în diagrama din figura 3.10*a*. În prima etapă considerăm diagrama din figura 3.10*b* din care extragem produsul $x_2 x_0$.

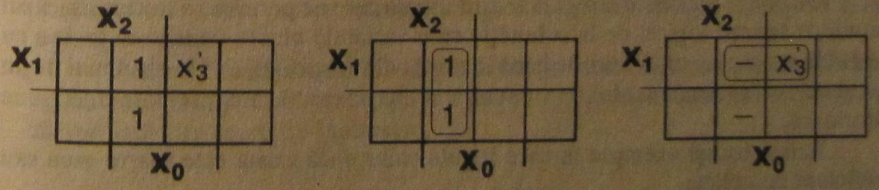



**Figura 3.10**

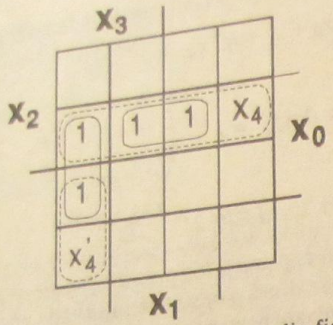



Figura 3.11

În a doua etapă considerăm diagrama din figura 3.10*c* din care extragem produsul $(x_1 x_0) x_3'$; în paranteză avînd produsul asociat suprafeţei înmulţit cu "valoarea", dă $x_1 x_0 x_3'$. (Produsul $x_2x_0$ a fost înmulţit cu *1*, valoarea suprafeţei, rămînînd astfel nemodificat.) Deci funcţia logică extrasă va fi

$$x_2 x_0 + x_1 x_0 x_3'. \square$$

**Exemplul 3.14** Funcţia definită în figura 3.11 este extrasă după ce s-au definit suprafeţele (cu linie continuă pentru prima etapă şi cu linie întreruptă pentru cea de-a doua) sub forma :

$$x_3 x_1' x_0 + x_2 x_1 x_0 + x_2 x_0 x_4 + x_3 x_2' x_1' x_4'. \square$$

Minimizarea unei funcţii logice este o etapă importantă în proiectarea sistemelor digitale dar trebuie întotdeauna urmată de *transformarea ecuaţiei* obţinute astfel încît să poată satisface şi unor restricţii tehnologice sau unora impuse de sistemul din care circuitul face parte.

### 3.1.5 Funcţii definite în limbaj natural

De multe ori aplicaţiile concrete ne pun în faţa unor probleme care necesită definirea unor funcţii cu multe variabile. Mult, poate să însemne şi un număr de *8* variabile. Într-adevăr, un tabel de adevăr pentru *8* variabile are *256* de rînduri. De asemenea, diagramele Veitch-Karnaugh devin inoperante pentru mai mult de *5, 6* variabile.

De multe ori există o regulă foarte simplă care ne permite să trecem, în cazul anumitor funcţii logice, de la o funcţie cu $n$ variabile binare de intrare la una cu $n+1$. Un exemplu, supărător de banal este cel al funcţiei *AND*. Un altul mai puţin banal este cel al sumatorului de numere de $n$ biţi pe care-l vom trata într-o secţiune ulterioară.

Există totuşi exemple în care regula chiar dacă există este foarte greu sau ineficient de aplicat.

**Exemplul 3.15** În figura 3.12 este reprezentat un *CLC* cu funcţia de *codificator prioritar* (priority encoder) care generează la ieşirile $O_2O_1O_0$ codul



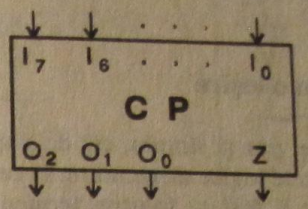



Figura 3.12

binar asociat intrării cu indice maxim pe care este aplicată valoarea *1*. Ieşirea $Z$ este activă cînd nici una din intrări nu este activată cu valoarea *1*. Astfel, dacă $I_7I_6..I_0 = 01011001$ atunci $Z = 0$ iar $O_2O_1O_0 = 110$. Aceeaşi ieşire avem şi pentru *01000000* aplicat pe intrări.

Definirea prin tabel a funcţiilor $O_2,O_1,O_0$ şi $Z$ este foarte nepractică. Vom încerca să le evidenţiem direct.

În consecinţă, vom observa că $O_2 = 1$ dacă una din intrările $I_7,...,I_4$ este activată. Deci

$$O_2 = I_7 + I_6 + I_5 + I_4.$$

Ieşirea $O_1$ este activă dacă $I_7$ sau $I_6$ au valoarea *1* sau dacă $I_3$ sau $I_2$ sînt active cu condiţia ca $I_5$ şi $I_4$ să fie *0*. Scriem deci

$$O_1 = I_7 + I_6 + (I_3 + I_2)I_4'I_5'.$$

Similar se poate raţiona pentru $O_0$ rezultînd

$$O_0 = I_7 + I_5I_6' + I_3I_6'I_4' + I_1I_6'I_4'I_2'.$$

Pentru ieşirea $Z$ este evident

$$Z = I_7'I_6'I_5'I_4'I_3'I_2'I_1'I_0'.$$

Circuitul poate avea şi o interpretare aritmetică. Dacă notăm cu $N$ numărul binar aplicat pe cele *8* intrări şi cu $M$ numărul binar de trei biţi de la ieşirile $O_2O_1O_0$, se poate scrie că

$$M = \lfloor log_2 N \rfloor.$$

Circuitul calculează partea întreagă a logaritmului în bază doi, iar ieşirea $Z$ indică operaţie eronată. □

În cazul circuitului *CP* există şi o definiţie recursivă care permite conceperea unuia cu $n$ intrări dacă se cunoaşte unul cu $n/2$ intrări. Cititorul o poate căuta, făcînd astfel un exerciţiu interesant.

## 3.2 Implementarea cu porţi a circuitelor combinaţionale

### 3.2.1 Circuite combinaţionale cu o ieşire

Termenul de *poartă logică* sau, pur şi simplu, cel de *poartă* s-a impus în limbajul curent referindu-se la circuitele logice elementare, în cea mai mare parte reprezentate în figura 3.2. Corect ar fi fost să folosim termenul numai pentru circuitul logic *AND* la care valoarea lui $x_1$ permite $(x_1 = 1)$ sau nu permite $(x_1 = 0)$ trecerea valorii lui $x_2$ la ieşire. Altfel spus, $x_1$ "deschide poarta" pentru $x_2$.

O funcţie logică poate fi implementată folosind reţele de porţi logice convenabil interconectate. Întotdeauna pentru o funcţie logică există mai multe soluţii acceptabile. Alegerea uneia se va face în funcţie de criterii suplimentare ce depăşesc simpla respectare a funcţionalităţii.

• **Timpul de propagare** a semnalului printr-o poartă logică, neglijat în prezentarea teoretică anterioară, devine important din perspectiva unei realizări concrete. În figura 3.13 am reprezentat două porţi deschise prin aplicarea valorii *1* pe cîte o intrare. Prin aceste porţi se poate propaga semnalul $x$ care evoluează în timp ca în figură, comutînd între nivele electrice corespunzătoare valorilor *1* şi *0*. Ieşirile $y_1$ şi $y_2$ reacţionează cu întîrzieri date de timpi de propagare $(t_p)$ specifici:

- $t_{pLH}$ - timpul de propagare asociat tranziţiei ieşirii între *0* (low, *L*) şi *1* (high, *H*);

- $t_{pHL}$ - timpul de propagare asociat tranziţiei ieşirii între *1* şi *0*.

Întotdeauna $t_{pLH} \neq t_{pHL}$ şi, de regulă, $t_{pLH} > t_{pHL}$. În tehnologiile standard este de ordinul nanosecundelor sau zecilor de ns.

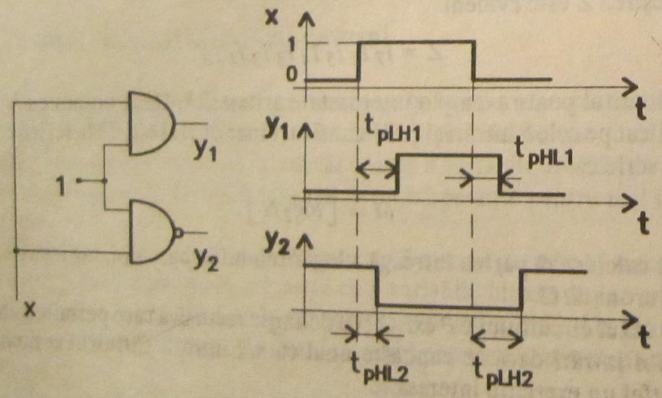



**Figura 3.13**



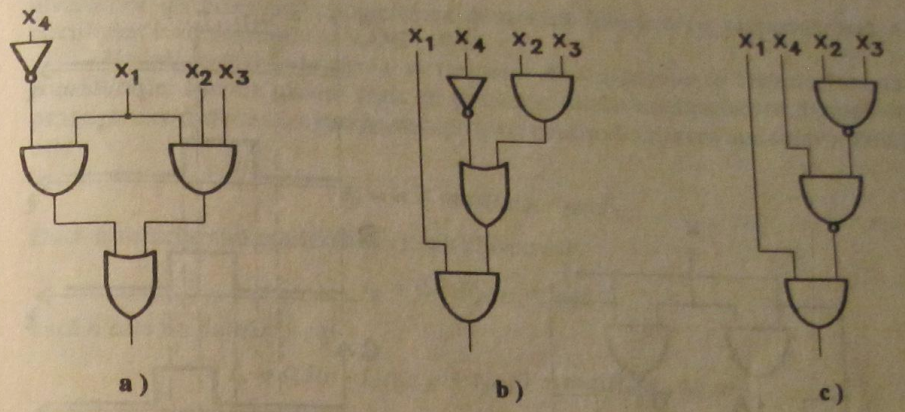



**Figura 3.14**

• **Fan-out**-ul unui circuit reprezintă *numărul maxim de circuite*, realizate în tehnologie identică, pe care un astfel de circuit le poate comanda fără ca semnalul la ieşirea sa să se alterneze astfel încît să-şi piardă semnificaţia. În tehnologii curente este *10*.

• **Fan-in**-ul unui circuit reprezintă numărul de intrări pe care acesta-l posedă.

**Exemplul 3.16** Vom lua în consideraţie funcţia logică din *Exemplul 3.9* care poate avea asociat circuitul reprezentat în figura 3.14*a*, în care pentru inversarea variabilei $x_4$ am folosit un circuit inversor *(NOT)*, pentru primul produs o poartă *AND* cu două intrări, iar pentru cel de al doilea produs o poartă *AND* cu trei intrări. Suma logică a fost realizată cu un circuit *OR* cu două intrări.

O transformare elementară a funcţiei oferă forma

$$x_1 (x_4' + x_2 x_3)$$

ce are asociat circuitul din figura 3.14*b*, care este mai simplu.

Aplicînd teorema lui De Morgan obţinem forma

$$x_1 (x_4 (x_2 x_3)')'$$

şi circuitul corespunzător în figura 3.14*c*.

Care este soluţia cea mai bună? Un răspuns absolut valabil în toate situaţiile nu poate fi dat la această întrebare. Dacă se restricţionează contextul atunci răspunsurile încep să fie posibile.

Ceea ce este însă foarte important este că posibilităţile de *transformare* ale funcţiilor logice trebuiesc la maximum folosite pentru ca spaţiul soluţiilor să fie cît mai larg, astfel încît alegerea să se poată face după cît mai multe eventuale criterii.

Evident, în exemplul anterior, soluţia cea mai bună este ultima. Presupune un număr minim de circuite ca fan-in minim. Dar nu numai atît!

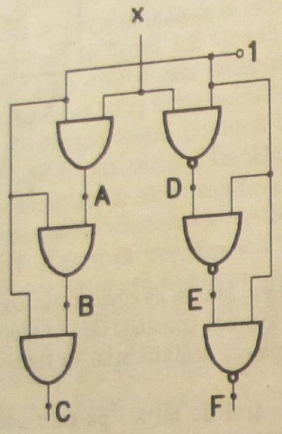



Figura 3.15

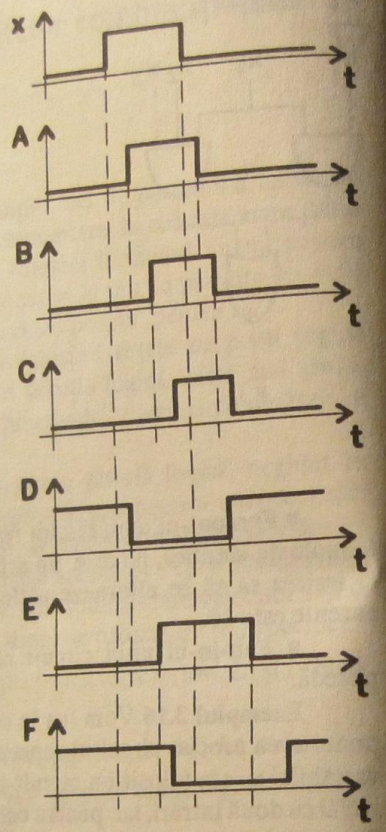



Figura 3.16

• **Utilizarea porţilor inversoare** în implementarea circuitelor combinaţionale este serios motivată de facilităţi tehnologice, uniformitate structurală şi *efecte de propagare*. Cel mai important este ultimul motiv.

Să considerăm circuitele din figura 3.15 şi formele de undă, asociate lor, din figura 3.16, ce au fost desenate în situaţia, curentă, în care $t_{pLH} > t_{pHL}$. Durata impulsului la intrarea $x$ este suficient de mică pentru ca efectele să fie evidente. Se observă tendinţa de scădere a duratei impulsului la trecerea prin circuite neinversoare. După un număr suficient de mare de niveluri, un impuls scurt poate să dispară ca semnal electric cu toate că, din punct de vedere logic, el ar trebui să existe. La viteze mari de lucru, unde apar impulsuri foarte scurte, un astfel de fenomen este foarte probabil. La trecerea prin circuitele *NAND* se observă tendinţa de refacere, după fiecare două nivele logice, a duratei impulsului, efectul de deformare în timp nefiind cumulativ. După un număr oricît de mare de porţi inversoare abaterea maximă poate fi dată, aproximativ, de diferenţa $t_{pLH} - t_{pHL}$. Cascadarea porţilor inversoare nu riscă, astfel, să determine dispariţia impulsurilor, ca în cazul anterior. Deformarea minimă introdusă de folosirea



nivelurilor de porţi logice inversoare determină folosirea cu preponderenţă a circuitelor inversoare de tip *NAND*, *NOR*.

Un alt motiv este acela al timpului de propagare ce trebuie luat în consideraţie. Într-un sistem logic ce posedă $n$ niveluri neinversoare, timpul de propagare ce trebuie luat în consideraţie (deci cel din situaţia cea mai defavorabilă) este

$$t_p = n \times max(t_{pLH}, t_{pHL}).$$

Dacă nivelurile sînt realizate cu circuite inversoare

$$t_p = 0{,}5n(t_{pLH} + t_{pHL})$$

dacă $n$ este un număr par şi

$$t_p = 0{,}5(n - 1)(t_{pLH} + t_{pHL}) + max(t_{pHL}, t_{pLH})$$

dacă $n$ este impar.

Avantajul utilizării circuitelor inversoare este suficient argumentat cu aceste două motive. La acestea se adaugă altele ce vor fi evidenţiate în alte secţiuni.

Orice funcţie logică, am văzut, este o *sumă de produse (FND)*. Aplicînd teorema lui De Morgan ea devine un *NAND* de *NAND*-uri, după cum urmează:

$$f_i^j = ((f_k^{j-n} x_{n-1} x_{n-2} \dots x_0)'(f_l^{j-n} x_{n-1} x_{n-2} \dots x_0)' \dots (f_m^{j-n} x_{n-1} x_{n-2} \dots x_0)')',$$

pornind de la *Definiţia 3.4*.

**Exemplul 3.17**

$$x_3'x_2x_1'x_0 + x_2'x_1 + x_3x_2 = ((x_3'x_2x_1'x_0)'(x_2'x_1)'(x_3x_2)')'. \ \square$$

**Exemplul 3.18** Reluînd *Exemplul 3.9* obţinem

$$x_1x_4' + x_1x_2x_3 = ((x_1x_4')'(x_1x_2x_3)')'$$

şi rezultă circuitul din figura 3.17, realizat cu circuite *NAND* cu o intrare, cu două şi cu trei intrări. □

Concluzionăm afirmînd că porţile inversoare conectate în cascadă *deformează minimal durata impulsurilor* ce se propagă prin ele şi introduc o *întîrziere minimă*. Folosirea lor cu preponderenţă va fi astfel justificată.

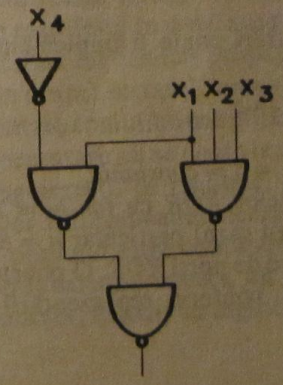



Figura 3.17

• **Complexitatea şi adîncimea** unui *CLC* caracterizează cantitativ această categorie de structuri.

Complexitatea notată cu $S(n)$, de la englezescul *size*, dă o imagine asupra mărimii circuitului, asupra efortului structural depus pentru realizarea lui. Cu $n$ se notează numărul variabilelor binare de intrare. $S(n)$ poate fi estimat în mai multe moduri nici unul nefiind incontestabil cel mai corect. În funcţie de tehnologia în care sînt realizate concret circuitele o definiţie sau alta poate fi mai corectă. Noi vom adopta următoarea definiţie.

**Definiţia 3.5** Notăm cu $S(n)$ asociat unui *CLC*, numărul de intrări al tuturor circuitelor prin care se configurează acel *CLC*. □

**Exemplul 3.19** Circuitul din figura 3.17 are complexitatea $S = 8$.

Această complexitate nu este dependentă de $n$ datorită naturii funcţiei. □

**Exemplul 3.20** Complexitatea circuitului *XOR* cu $n$ intrării realizat pornind de la *FND* este (conform *Propoziţia3.11*)

$$S_{XOR}(n) = n2^{n-1} + 2^{n-1} + n,$$

unde primul termen corespunde celor $2^{n-1}$ mintermeni, cel de al doilea circuitului ce-i sumează, iar ultimul inversoarelor pentru cele $n$ variabile binare de intrare. □

În studiul tehnologiilor *VLSI* (*V*ery *L*arge *S*cale *I*ntegation) complexitatea este definită în funcţie de *aria* ocupată de circuit pe cipul de siliciu. Nu va trebui să ne mirăm dacă astfel se vor găsi rezultate calitativ distincte, deoarece, în acest caz, un rol important îl au conexiunile dintre circuite, conexiuni, ce în anumite aplicaţii, pot avea un rol preponderent.

**Definiţia 3.6** Notăm cu $D(n)$ adîncimea (*d*epth) unui *CLC*, înţelegînd prin aceasta numărul maxim de niveluri logice (porţi logice) pe care-l parcurge pînă la ieşire un semnal aplicat la intrare. □

**Exemplul 3.21** Circuitul din figura 3.17 are adîncimea $D = 3$, ce este dată de numărul de circuite parcurs de variabila $x_4$. □

**Propoziţia 3.12** Adîncimea minimă ce poate fi asociată, în cazul general, unui *CLC* este $D = 3$. □

*Demonstraţia* este evidentă pornind de la analiza *FND*, care implică un *OR*, nişte *AND*-uri şi inversoare pentru calculul variabilelor ce intervin inversate. Am văzut că cele trei niveluri pot fi realizate şi cu acelaşi tip de circuit: *NAND*-ul. □

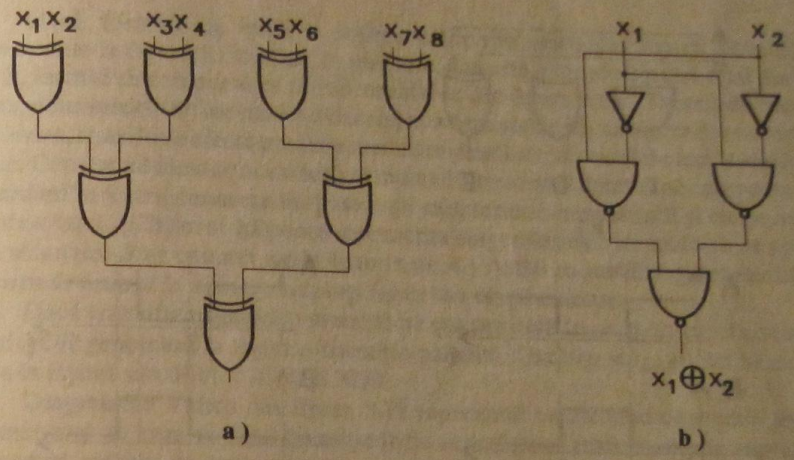
De regulă un *circuit realizat cu adîncime minimă tinde către o complexitate maximă.*

**Exemplul 3.22** Circuitul *XOR* cu $n$ intrări poate fi implementat în două maniere. Odată în *FND*, caz în care

$$S_{XOR}(n) = (n+1)2^{n-1} + n$$

conform *Exemplu 3.20* iar adîncimea este $D(n) = 3 = constantă$.
O altă variantă ar putea fi utilizarea circuitelor *XOR* cu *fan-in* $= 2$ conectate conform regulii sugerate în figura 3.18*a* pentru cazul particular $n = 8$. Structura arborescentă conţine $n-1$ circuite şi $\log_2 n$ nivele de porţi *XOR*. O poartă *XOR* este caracterizată prin $S_{XOR}(2) = 8$, $D_{XOR}(2) = 3$, conform reprezentării din figura 3.18*b*.






a) b)

**Figura 3.18**

Deci într-o structură arborescentă

$$S_{XOR}(n) = 8(n-1),$$

$$D_{XOR}(n) = 3\log_2 n.$$

Deci preţul plătit pentru o adîncime mică (deci viteză mare de execuţie) este creşterea dramatică a complexităţii. □

Adîncimea circuitului este, în expunerea noastră, proporţională cu timpul de propagare prin circuit. În multe aplicaţii concrete, realizate cu circuite rapide dar cu conexiuni relativ lungi, această presupoziţie nu mai este valabilă. Întîrzierile datorate conexiunilor pot deveni semnificative impunînd o analiză mai nuanţată. Pentru circuite realizate în tehnologii *VLSI* sau în sisteme de mare complexitate, efectul conexiunilor asupra timpului de propagare nu mai poate fi neglijat.

• **Hazardul** este consecinţa faptului real că în *comutarea* a două variabile binare nu se poate controla în nici un fel sincronizarea lor absolută şi în *propagare* nu pot fi controlate întîrzierile decît ca valori extreme, nu şi ca valori efective. Hazardul se manifestă în două moduri distincte. Unul corespunde comutării ce nu poate fi sincronă iar celălalt propagărilor diferenţiate prin reţele de porţi, fapt ce introduce, în ultimă instanţă, tot un asincronism. Semnalele $A$ şi $B$ din figura 3.19*b* sînt reprezentate comutînd în antifază şi *sincron*. Sincronismul teoretic presupus de această reprezentare nu se poate realiza concret decît accidental şi necontrolat prin procedee inginereşti. În realitate, chiar cînd tindem să obţinem un sincronism, semnalele vor fi întotdeauna decalate (figura 3.19*c*). Sincronismul presupune un interval de timp *nul* între comutarea celor două semnale. Ori noi ştim că zeroul, chiar dacă o fi existînd uneori, nu poate fi controlat, impus sau sesizat printr-un procedeu concret. Pentru un inginer *zero nu există* decît ca un model teoretic.

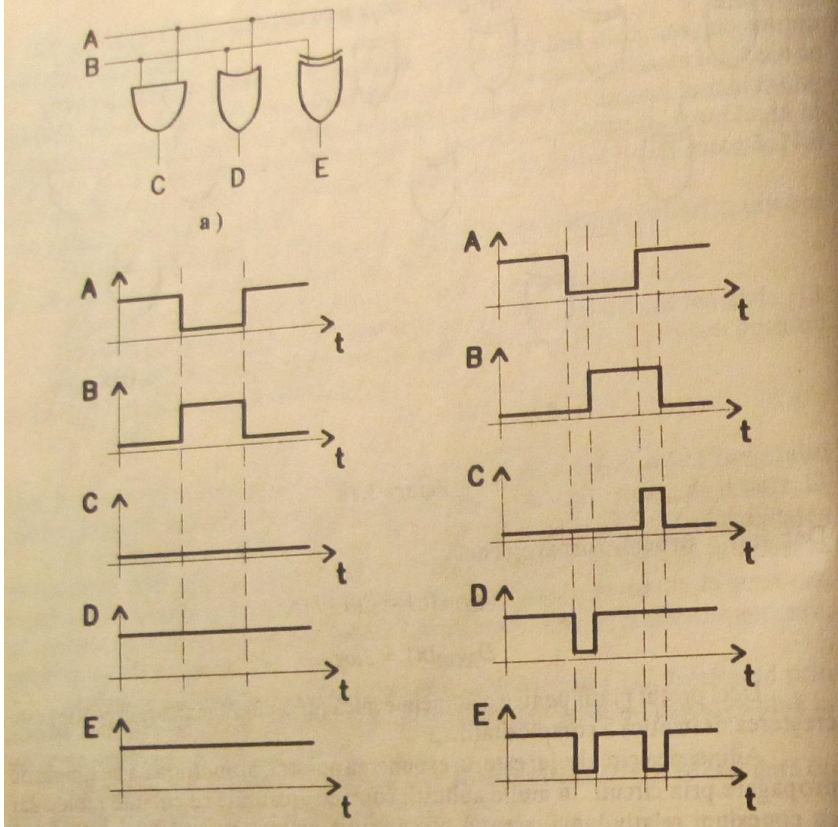



Figura 3.19



Ideal, formele de undă la ieşirea celor trei porţi din figura 3.19*a* sînt reprezentate în figura 3.19*b*. Dar, în realitate, datorită asincronismului fatal dintre *A* şi *B*, ieşirile se comportă ca în reprezentarea din figura 3.19*c*. Tranziţiile foarte scurte, determinate de inevitabilul decalaj al momentelor în care *intrările comută în sens invers*, constituie efecte parazite, nedorite şi neluate în seamă de teoria algebrei logice. Caracterul aleator al asincronismului dintre *A* şi *B* determină comportarea "*hazardată*" a ieşirii deoarece nu putem şti care semnal comută întîi şi cît de mare este decalajul. (Cititorul îşi poate reprezenta singur formele de undă ce ar apare dacă semnalul *B* ar comuta puţin înainte de *A*.) Astfel se justifică şi termenul de *fenomen de hazard la ieşirea unei porţi logice* sau simplu *hazard*.

Dacă tranziţia negativă o precede pe cea pozitivă ($t_L < t_H$), atunci circuitul *OR* şi *XOR* generează la ieşire o tranziţie parazită. Dacă $t_H < t_L$, atunci hazardul apare la ieşirea circuitelor *AND* şi *XOR*.

Diagramele Veitch din figura 3.19 reprezintă un alt mod de evidenţiere a fenomenului de hazard. Cînd tranziţia între suprafeţe se realizează prin suprafeţe cu aceeaşi valoare ca cele iniţiale şi finale, fenomenul de hazard nu se poate manifesta. Cum tranziţia pe diagonală se poate numai teoretic imagina hazardul este posibil la *AND* şi *OR* şi este întotdeauna prezent la *comutarea ambelor intrări* în cazul *XOR*-ului.

**Propoziţia 3.13** Dacă, la intrarea unui *CLC* comută cel puţin semnalul aplicat pe două intrări, atunci este posibilă apariţia fenomenului de hazard. □

Acest hazard este datorat asincronismului (inevitabil) al comutării intrărilor.

Poate fi pus în evidenţă şi un alt tip de hazard: cel datorat *propagării efectului unui semnal* printr-un *CLC*. Să considerăm circuitul din figura 3.20.

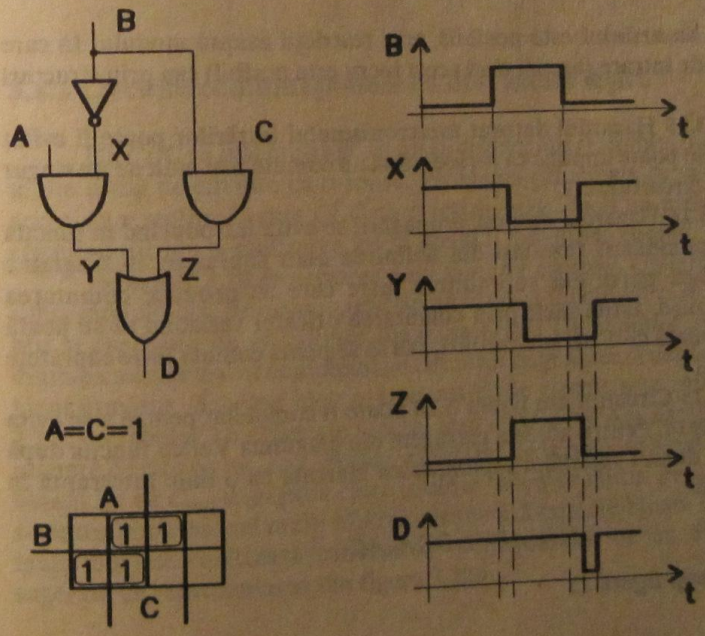



Figura 3.20

Conform funcţiei logice asociate

$$D = AB' + CB,$$

dacă $A = C = 1$, atunci pentru orice valoare a lui $B$ ieşirea $D$ nu comută rămînînd egală cu $1$. Acest lucru este valabil numai teoretic, pentru circuite ideale, cu timp de propagare nul. În realitate, formele de undă ce caracterizează funcţionarea circuitului sînt cele din figura 3.20, ce pun în evidenţă tranziţia parazită a ieşirii. Efectul comutării lui $B$ ajunge cu întîrzieri diferite la ieşire (porţile *AND* nu sînt identice şi întîrzierea dată de inversor este diferită de zero).

Diagrama Veitch din aceeaşi figură ne arată că prin *comutarea unei singure variabile* poate fi părăsită o suprafaţă şi se trece în altă suprafaţă cu aceeaşi valoare logică egală cu $1$. Această variabilă este $B$. Prin comutarea variabilei $A$ putem sări din suprafaţe de $1$-uri în suprafeţe de $0$-uri şi invers. La fel pentru $C$. În acest caz ieşirea comută net din $1$ în $0$ sau invers.

**Propoziţia 3.14** Dacă prin comutarea unei singure variabile se poate sări dintr-o suprafaţă de $1$-uri în alta tot de $1$-uri, atunci circuitul caracterizat prin această diagramă Veitch poate genera la ieşire tranziţii parazite (hazard). □

Am folosit în propoziţia anterioară cuvîntul "poate" pentru a lua în consideraţie şi echilibrările *accidentale* ale timpilor de propagare pe diferitele căi pe care efectul *unei* tranziţii afectează ieşirea.

Observăm că am putea asimila *hazardul de propagare* cu cel dat de asincronism, dacă luăm în consideraţie în figura 3.20 circuitul cu intrările $A,B,C$ şi $X$. Intrările $B$ şi $X$ comută în contratimp şi asincron. Deci, în ultimă instanţă, avem de a face cu un singur tip de hazard, *cauza lui unică fiind asincronismul* (care o dată se manifestă la bornele sistemului iar altă dată în interiorul lui). Vom menţine în continuare distincţia între cele două tipuri de hazard datorită metodelor distincte prin care-l evităm.

• **Evitarea hazardului** este posibilă prin restricţii asupra modului în care variază semnalele de intrare (atunci cînd acest lucru este posibil) sau prin structuri redundante.

**Propoziţia 3.15** Hazardul datorat asincronismului intrărilor poate fi evitat numai atunci cînd se poate impune ca niciodată să nu comute mai mult de un singur semnal la intrarea circuitului. □

**Propoziţia 3.16** Hazardul datorat propagării se evită introducînd în funcţia logică termeni suplimentari rezultaţi din definirea unor suprafeţe, în diagrama Veitch, care să facă parte din suprafeţele între care se produce comutarea generatoare de hazard, astfel încît prin comutarea oricărei variabile să se poată rămîne într-o suprafaţă de 1-uri sau zerouri, sau să se poată comuta între suprafeţe cu valori diferite. □

**Exemplul 3.23** Circuitul din figura 3.20 poate fi completat pentru a se evita hazardul de propagare. Pentru aceasta extragem din diagrama Veitch funcţia după ce în prealabil am definit *suprafaţa suplimentară* marcată cu o linie întreruptă în figura 3.21. Rezultă

$$D = AB' + BC + AC$$

şi circuitul din aceeaşi figură. □



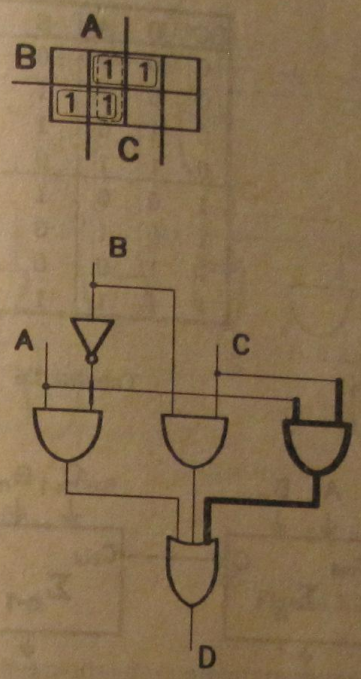


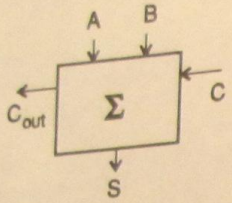

**Figura 3.21**

*Propoziţia 3.15* oferă premizele evitării hazardului iar *Propoziţia 3.16* garantează absenţa hazardului. Prima este o condiţie absolut necesară, fără de care hazardul nu poate fi evitat.

## 3.2.2 Circuite combinaţionale cu mai multe ieşiri

Un *CLC* cu mai multe ieşiri nu se obţine, pur şi simplu, punînd împreună mai multe astfel de circuite cu o ieşire. Problema principală ce se cere rezolvată este aceea de a realiza fiecare *CLC* cu o ieşire, astfel încît un număr cît mai mare de circuite utilizate la realizarea lor să poată fi folosite în comun.

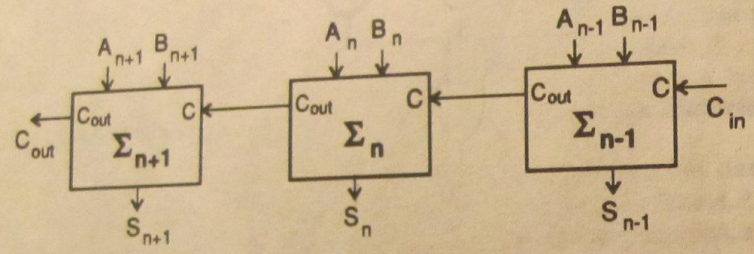
**Exemplul 3.24** *Proiectarea unui sumator complet*. Un circuit *XOR* realizează suma modulo doi. *Semisumatorul* de un bit este un circuit care adaugă sumei a doi biţi şi *depăşirea* de la ordinul binar anterior. Un *sumator complet* este format dintr-un semisumator completat cu circuitul ce generează depăşirea pentru ordinul binar următor. Schema bloc a unui astfel de circuit este reprezentată în figura 3.22*a* iar tabelul 3.6 defineşte funcţia logică asociată. Am notat cu $A$ şi $B$ cele două numere de cîte un bit, cu $C$ depăşirea (carry) de la ordinul binar anterior, cu $S$ suma binară iar cu $C_{out}$ depăşirea către ordinul binar următor. Pentru construirea unui sumator binar de mai mulţi biţi (care sumează numere binare reprezentate pe mai mulţi biţi) se realizează conectarea sumatoarelor de un bit conform regulii sugerate de reprezentarea din figura 3.22*b* .




a)

| C | B | A | S | Cout |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

**Tabelul 3.6**




b)

**Figura 3.22**

Ecuaţiile ce descriu circuitul $\Sigma$ sînt următoarele :

$$S = A \oplus B \oplus C,$$

$$C_{out} = AB + AC + BC$$

ele fiind deduse din tabelul 3.6 şi *minimizate fiecare în parte*. Dar, nu s-ar putea pune problema *minimizării ansamblului* celor două funcţii, cu riscul de a complica pe una din ele? În acest caz se poate. Vom scrie

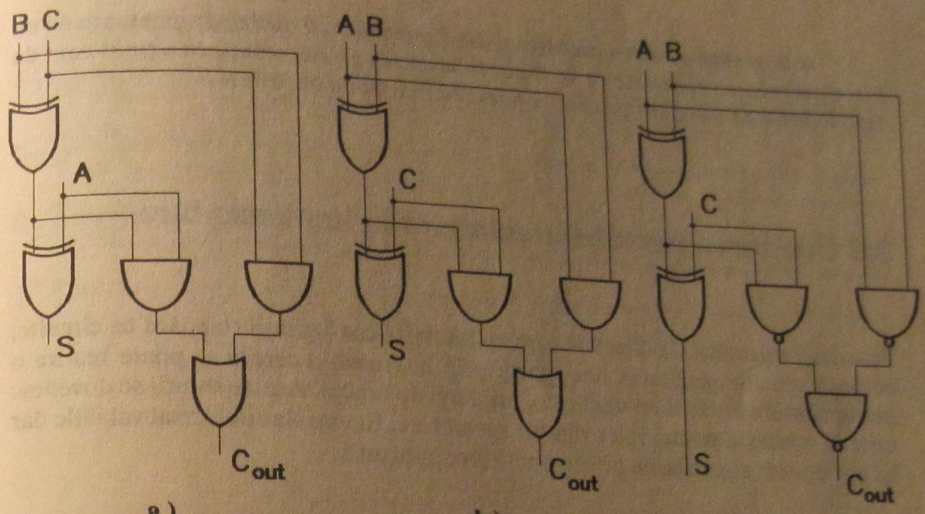
$$C_{out} = A(B+C) + BC$$

şi observăm (cum?) că s-ar putea ca

$$C_{out} = A(B \oplus C) + BC$$

ceea ce se demonstrează imediat. Deci, circuitul *XOR* pentru $B \oplus C$ *poate fi folosit pentru implementarea ambelor funcţii, minimizîndu-se astfel efortul global.* Rezultă circuitul din figura 3.23.






a) b) c)

**Figura 3.23**

Analizînd modul de configurare al unui sumator de mai mulţi biţi observăm faptul că timpul de propagare este critic pentru semnalul $C$ deoarece el provine de la un circuit anterior şi apare, prin aceasta, întîrziat la intrarea sumatorului $\Sigma_i$. Ultima variabilă ce apare va trebui să se propage pe calea cea mai rapidă. Observăm că în implementarea realizată variabila $C$ este întîrziată de un *XOR* un *AND* şi un *OR*, parcurgînd cel mai lung traseu. Nu am putea rescrie ecuaţiile astfel încît să-i oferim o cale mai scurtă? Simetria avansată a ecuaţiilor permite şi forma

$$S = (A \oplus B) \oplus C,$$

$$C_{out} = C(A \oplus B) + AB,$$

ce asigură, pe calea de propagare a transportului, tranziţia semnalului numai printr-un *AND* şi un *OR*. Rezultă aceeaşi schemă (figura 3.23*b*) cu o altă notare a intrărilor. Printr-o simplă renotare obţinem o creştere notabilă a vitezei!

Tot din modul de interconctare al sumatoarelor remarcăm faptul că semnalul de depăşire se propagă printr-un număr de $2n$ porţi neinversoare, în cazul unui sumator de $n$ biţi. Efectul poate fi dezastruos în funcţionare la mare viteză, conform comentariului anterior făcut (paragraful 3.2.1). Evităm aceste efecte aplicînd teorema lui De Morgan, obţinînd

$$C_{out} = ((C(A \oplus B))'(AB)')'$$

şi circuitul corespunzător reprezentat în figura 3.23*c*.

Trebuie să remarcăm cît de lungă este calea de la expresia minimizată la o soluţie concretă. □

• Sugerăm reluarea problemei din *Exemplul 3.10* utilizînd sumatoare de un bit. Compararea rezultatelor va fi net în favoarea primei soluţii ce a ţinut cont de restricţionarea valorilor pe care le poate lua variabila de intrare.

## 3.3 Utilizarea circuitelor standard în implementare

Practica proiectării *CLC*-urilor a impus anumite configuraţii standard de circuite, ce depăşesc complexitatea unei porţi, prin intermediul cărora se poate realiza o implementare eficientă a unei categorii largi de funcţii. Aceste soluţii se dovedesc utile în contexte tehnologice riguros specificate. Ele nu sînt universal valabile dar îşi pot dovedi eficacitatea printr-o utilizare oportună.

### 3.3.1 Circuite implementate cu multiplexoare

Ne putem pune întrebarea dacă există sau nu un circuit universal pentru implementarea oricărei funcţii logice. Răspunsul este afirmativ şi se bazează pe *Definiţia 3.4* care poate avea asociat circuitul din figura 3.24, a cărui funcţie logică este

$$W = I_{2^n-1} S_{n-1} S_{n-2} \dots S_0 + I_{2^n-2} S_{n-1} S_{n-2} \dots S'_0 + \dots + I_0 S'_{n-1} S'_{n-2} \dots S'_0 .$$

Prin configuraţiile binare $S_{n-1} \dots S_0$ este *selectată* la ieşire valoarea de pe una din intrările $I_i$. În cazul particular $n = 3$ rezultă funcţia

$$W = (I_7 S_2 S_1 S_0 + I_6 S_2 S_1 S_0' + I_5 S_2 S_1' S_0 + I_4 S_2 S_1' S_0' + I_3 S_2' S_1 S_0 + I_2 S_2' S_1 S_0' + I_1 S_2' S_1' S_0 + I_0 S_2' S_1' S_0')E'$$

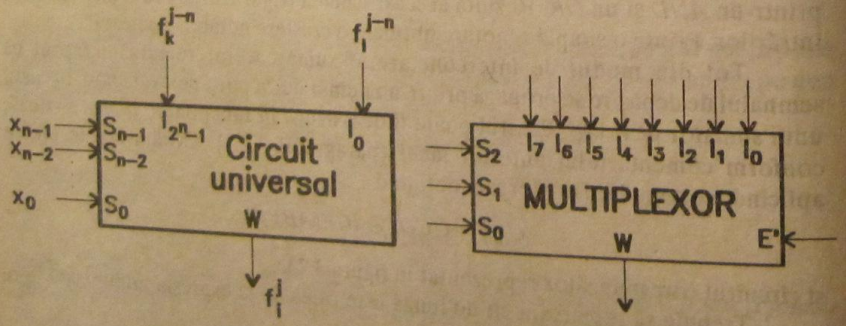



Figura 3.24




Figura 3.25



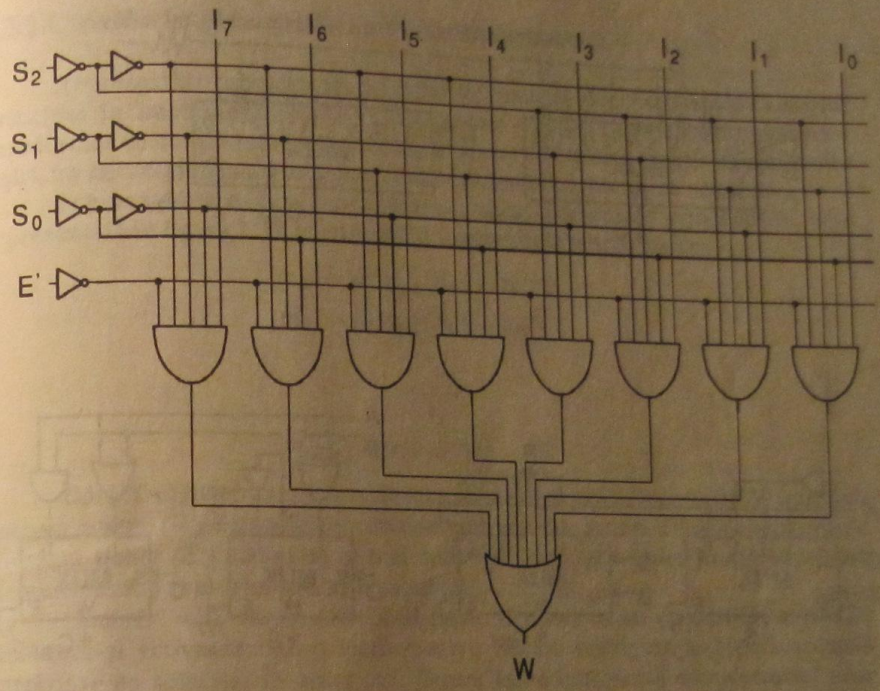



Figura 3.26

ce are asociat circuitul din figura 3.25 cu structura internă reprezentată în figura 3.26. Intrarea suplimentară $E'$ (enable) permite activarea $(E' = 0)$ sau inactivarea $(E' = 1)$ comandată a circuitului. Acest circuit îndeplineşte şi funcţia de *multiplexare*, motiv pentru care-l vom nota curent prin *MUX*. Inversoarele de pe intrările de selecţie şi de pe intrarea de validare a funcţionării $(E')$ au fost introduse pentru a nu creşte fan-out-ul circuitelor ce comandă aceste intrări.

**Exemplul 3.25** Implementarea unui sumator complet cu ajutorul a două *MUX*-uri se poate face în mai multe moduri funcţie de $n$ (*Definiţia 3.4*). Soluţia pentru $n = 3$ este redată în figura 3.27*a* unde pe intrările de tip $I_i$ au fost aplicate direct valorile ce rezultă din tabelul de adevăr (tabelul 3.6). Tot pornind de la tabelul de adevăr se construieşte şi soluţia pentru $n = 2$ aplicîndu-se pe aceleaşi intrări funcţii de o variabilă $(f_i^2)$ ce se determină grupînd în ordine cîte două linii din tabelul 3.6. Astfel, pentru $C = B = 0$ (primele două linii în tabel) se observă că $S = A$, pentru $C = 0$ şi $B = 1$ (următoarele două linii) $S = A'$ ş.a.m.d. Rezultă circuitul din figura 3.27*b* realizat cu două *MUX*-uri de patru căi. (Atenţie! Pentru o altă manieră de interconectare se pot obţine performanţe de viteză mai bune în cazul propagării transportului, conform discuţiei făcute în *Exemplul 3.24*. Cititorul va face această modificare). Adăugînd un singur inversor am redus dimensiunea *MUX*-urilor la jumătate! Merită să continuăm procesul?

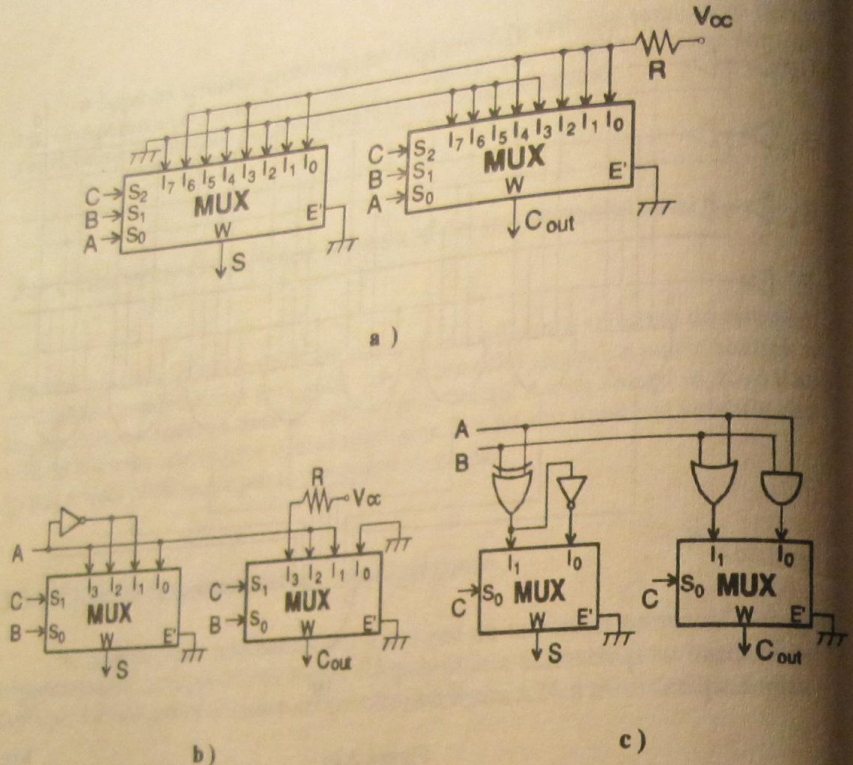



Figura 3.27

Pentru $n = 1$ soluţia este reprezentată în figura 3.27*c* şi a fost obţinută observînd că pentru $C = 0, S = B \oplus A$ şi $C_{out} = BA$ iar pentru $C = 1, S = (B \oplus A)'$ iar $C_{out} = B + A$.

(Notăm faptul că valoarea *0* poate fi generată conectînd intrarea la masă iar valoarea *1 conectînd printr-o rezistenţă* intrarea la alimentarea circuitului, $V_{cc}$.)

Remarcăm faptul că soluţia cea mai complexă este cea mai rapidă iar cea mai puţin complexă este cea mai lentă. Vom opta pentru una din cele trei în funcţie de contextul în care circuitul este folosit. Amintim faptul că pentru această funcţie există circuite specializate care asigură eficienţa maximă. □

Trebuie remarcat faptul că pentru *CLC*-uri cu mai multe ieşiri utilizarea acestei metode se poate dovedi deseori ineficientă datorită faptului că sînt calculaţi mintermenii de atîtea ori cîte funcţii implementăm. Într-adevăr, conform reprezentării detaliate în figura 3.26, fiecărui mintermen îi este asociat un *AND* în fiecare *MUX*. Un mintermen odată calculat nu poate fi folosit la realizarea mai multor funcţii. Această deficienţă este surmontată în secţiunea următoare.



### 3.3.2 Circuite implementate cu demultiplexoare

Reprezentare funcţiilor logice prin *FND* pune în evidenţă faptul că efortul principal în implementarea unui circuit este dat de calculul mintermenilor (produselor ce conţin toate variabilele de intrare). Pentru o funcţie cu mai multe ieşiri, un mintermen odată calculat, poate fi folosit în construirea oricăreia.

Schema bloc a unui *demultiplexor*, notat prescurtat prin *DMUX*, este reprezentată în figura 3.28, ieşirile avînd următoarele funcţii logice

$$m_0' = (x' x_{n-1}' x_{n-2}' \dots x_0')',$$
$$m_1' = (x' x_{n-1}' x_{n-2}' \dots x_0)',$$
$$\vdots$$
$$m'_{2^n-1} = (x' x_{n-1} x_{n-2} \dots x_0)'.$$

*DMUX*-ul îndeplineşte funcţia inversă multiplexării. Dacă un *MUX* transferă într-un punct ($W$) semnale din diverse puncte ($I_i$), atunci *DMUX*-ul împrăştie dintr-un punct ($E'$) semnalul in mai multe puncte ($O_i'$). Calea multiplexată sau demultiplexată este selectată prin codul ($S_{n-1}, \dots, S_0$).

Dacă $E' = 0$, atunci *DMUX*-ul poate fi privit ca un decodificator (*DCD*) deoarece-şi activează cîte o ieşire pentru fiecare configuraţie binară aplicată intrărilor de selecţie. În acest caz, *fiecare ieşire calculează complementul unui mintermen*.

Exemplificăm această structură printr-un *DMUX* pe patru căi reprezentat în figura 3.29. Intrarea $E'$ permite (enable) activarea ieşirilor ce în starea inactivă stau

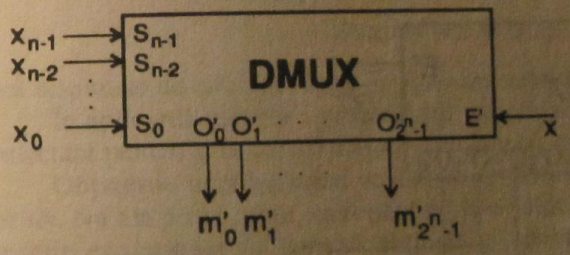



Figura 3.28

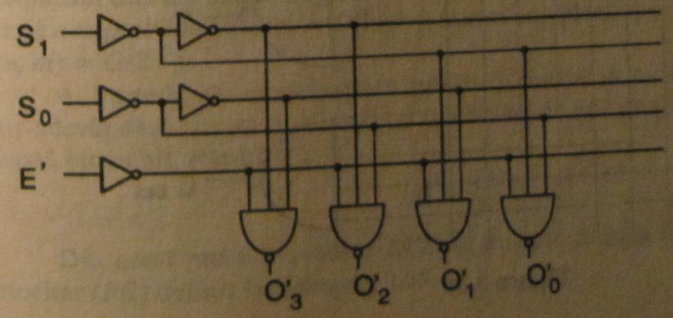



Figura 3.29

pe *1*. De ce este realizat acest circuit ca o structură inversoare? Din aceleaşi motive pentru care utilizăm preferenţial porţi inversoare. Am văzut că, aplicînd unei *FND* teorema lui De Morgan, obţinem posibilitatea implementării oricărei funcţii logice cu ajutorul *NAND*-urilor. Mintermenii negaţi sînt "sumaţi" tot printr-un *NAND*. Pe această observaţie se bazează utilizarea *DMUX*-urilor în realizarea circuitelor logice.

**Exemplul 3.26** Vom relua şi prin acest procedeu implementarea sumatorului complet cu un *DMUX* pe opt căi pentru care, dacă $E' = 0$

$$m_0' = (S_2'S_1'S_0')',$$
$$m_1' = (S_2'S_1'S_0)',$$
$$\vdots$$
$$m_7' = (S_2 S_1 S_0)'.$$

Avînd la dispoziţie mintermenii negaţi, cu două *NAND*-uri cu patru intrări, putem construi sumatorul complet conform figurii 3.30. La intrările porţilor au fost conectate produsele marcate cu *1* în tabelul de adevăr. □

Realizarea proiectării este foarte simplă. Ca şi în cazul folosirii *MUX*-urilor *nu presupune un proces de minimizare*, ci se reduce la un desen care se face inspectînd, simplu, tabelul de adevăr.

**Exemplul 3.27** pentru implementarea funcţiilor din tabelul 3.7 se impune, conform procedeului expus, folosirea, unui *DMUX*, a unei porţi cu *6* intrări (pentru $y_1$) şi a uneia cu *fan-in* = *2* (pentru $y_0$). Dacă am implementa însă funcţia $y_1'$, pe care apoi am nega-o, am putea folosi un *NAND* cu *fan-in* = *2* şi un inversor sau numai o poartă *AND*. □

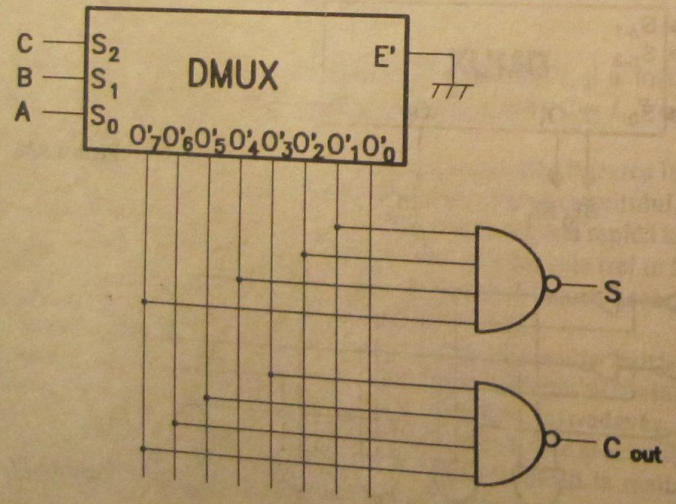



**Figura 3.30**



| $x_2$ | $x_1$ | $x_0$ | $y_1$ | $y_0$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |

**Tabelul 3.7**

În consecinţă:

- dacă în tabelul de adevăr funcţia este definită cu mai puţine *1*-uri decît *0*-uri, atunci vom folosi un *NAND* conectînd la intrările lui mintermenii marcaţi cu *1*;

- dacă în tabelul de adevăr funcţia este definită cu mai puţine *0*-uri decît *1*-uri, atunci vom implementa negatul funcţiei negate conectînd la intrarea unui *AND* ieşirile DMUX-ului corespunzătoare mintermenilor marcaţi cu *0*.

Astfel pentru o funcţie cu $n$ intrări vom folosi pe lîngă *DMUX* porţi cu cel mult $2^{n-1}$ intrări.

**Propoziţia 3.17** Orice circuit cu $n$ intrări şi $m$ ieşiri poate fi implementat cu un $DMUX$ cu $n$ intrări ( sau pe $2^n$ ieşiri ) la care se adaugă $m$ porţi cu $fan\text{-}in_{max} = 2^{n-1}$. □

Prin acest procedeu se obţine

$$S(n, m) = n + n2^n + m2^{n-1} = O(n2^n),$$
$$D(n, m) = 3 = O(1)$$

dacă dispunem de circuite cu *fan-in* şi *fan-aut* nelimitat.

În aplicarea corectă a procedeului, pentru $m$ mare, trebuie să fim atenţi la proiectare pentru a nu depăşi *fan-out*-ului asociat ieşirilor DMUX-ului.

Obţinerea unei adîncimi constante se instituie într-un rezultat mult prea pozitiv. Nu am putea, oare, ca renunţînd la o adîncime aşa de mică, la un timp de execuţie excepţional de bun, să obţinem o scădere a complexităţii circuitului? Răspunsul este afirmativ. Sugerăm cititorului să încerce să arate că dacă adîncimea creşte *acceptabil* la $D(n,m) = O(\log n)$, atunci complexitatea poate scade *drastic* la $S(n, m) = O(2^n)$.

- **Funcţia de exponenţiere** poate fi realizată cu ajutorul unui *DCD*. Într-adevăr dacă notăm cu $N$ configuraţia binară de intrare şi cu $M$ pe cea de ieşire, negată bit cu bit, rezultă că

$$M = 2^N.$$

Din acest punct de vedere *DCD*-ul execută funcţia inversă codificatorului prioritar *(PE)* definit în *Exemplul 3.15*.

Sugerăm cititorului să caute o soluţie, *cît mai simplă*, pentru calculul mărimii

$$M = 2^{2^N},$$

iar apoi să-i compare complexitatea cu a circuitului ce realizează numai

$$M = 2^N.$$

### 3.3.3 Utilizarea multiplexoarelor şi demultiplexoarelor în implementarea circuitelor combinaţionale.

Aşa cum am văzut în *Definiţia 3.4*, orice funcţie de $j$ variabile se implementează cu un *MUX* cu $n$ intrări de *selecţie* şi *mai multe* funcţii de $j$-$n$ variabile. Pentru funcţiile de aceleaşi $j$-$n$ variabile vom putea folosi un *DMUX* şi mai multe porţi. Rezultă o metodă ce combină pe cele două anterior expuse, utilă pentru funcţiile de mai multe variabile. Acest procedeu corespunde reprezentărilor cu funcţii în tabel sau diagramă. Să începem printr-un exemplu.

**Exemplul 3.28** În diagrama Veitch din figura 3.31 este definită o funcţie logică de 5 variabile. La nivelul variabilelor *A* şi *B* vom utiliza un *MUX* iar pentru cele patru funcţii de variabile *C*, *D* şi *E* vom folosi un *DMUX* şi porţi *AND* sau *NAND*.

Pentru implementarea cu *DMUX* vom face ca toate funcţiile să fie scrise sub *FND*. În consecinţă:

$$C + DE = C(D'E' + D'E + DE' + DE) + DE(C + C') =$$

$$= CD'E' + CD'E + CDE' + CDE + CDE + C'DE =$$

$$= (C'D'E' + C'D'E + C'DE')',$$

$$C + DE' + D'E = CD'E' + CD'E + CDE' + CDE + CDE' + C'DE' + CD'E +$$

$$+ C'D'E = (C'D'E' + C'DE)'.$$

Celelalte două funcţii fiind date în *FND*, concluzionăm că vor fi folosite: cîte un *AND* şi un *NAND* cu *fan-in* = 2 şi cîte un *AND* şi un *NAND* cu *fan-in* = 3, rezultînd schema din figura 3.32. □

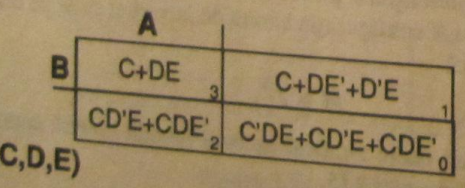



**Figura 3.31**



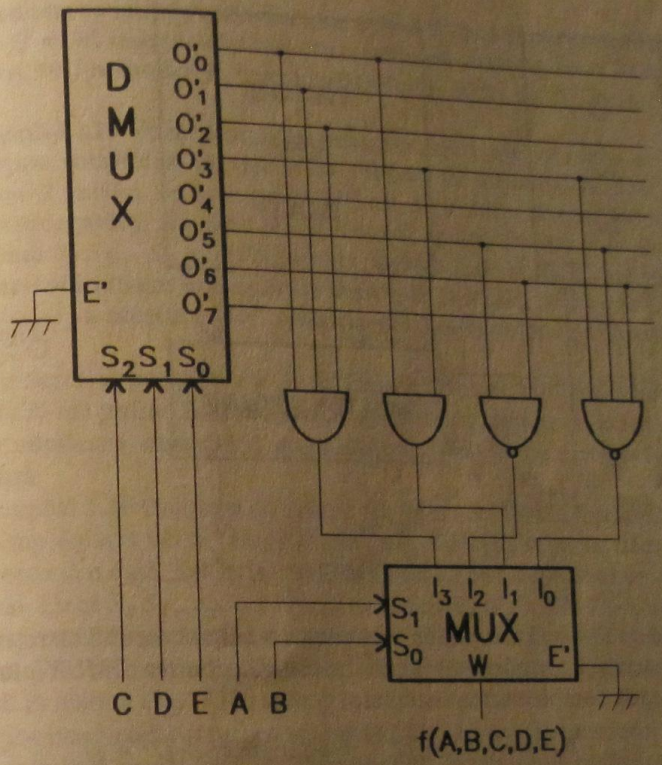



**Figura 3.32**

Structura generală ce poate fi propusă este cea din figura 3.33, ce conţine un *DCD*, o serie de *MUX*-uri (cîte unul pentru fiecare ieşire) şi o reţea de porţi interpusă. Particularizarea structurii se face prin conectarea convenabilă a porţilor. Această structură conţine două tipuri de circuite standard, *DMUX (DCD)* şi *MUX*-uri, la care se adaugă o reţea "programabilă" de porţi. "Programarea" porţilor se referă la conectarea lor fizică astfel încît să fie îndeplinită restricţia dată de fiecare definiţie particulară. Sugestia va sta la baza următoarei secţiuni.

- **Complexitatea** acestei structuri se calculează după cum urmează:

- reţeaua de porţi este formată din $m\ 2^j$ porţi (cîte una pentru fiecare intrare a celor $m$ *MUX*-uri) ce au maximum $2^{n-j-1}$ intrări (conform secţiunii anterioare);

- *DCD* este format din $2^{n-j}$ porţi cu $n$-$j$ intrări şi din $n$-$j$ inversoare;

- *MUX*-urile sînt în număr de $m$, sînt formate dintr-un *OR* cu $2^j$ intrări şi $2^j$ *AND*-uri cu $j$ intrări, la care se adaugă $j$ inversoare.

Rezultă:

$$S(n, m, j) = m2^{n-1} + (n-j)2^{n-j} + n-j + m(2^j + j2^j + j).$$

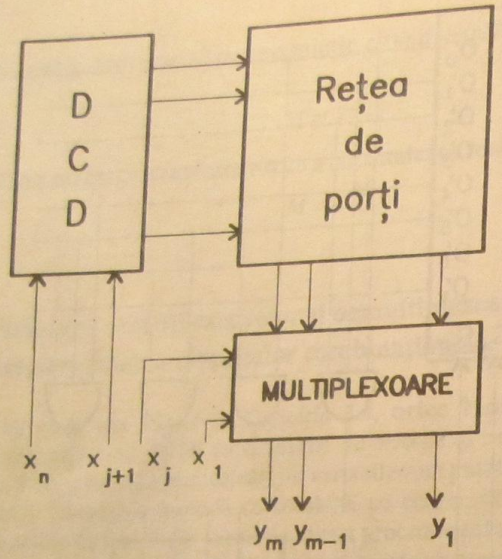



**Figura 3.33**

Comparind cu $S(n,m)$ calculată în secțiunea anterioară rezultă că rețeaua de porți rămîne de aceeași complexitate $(m2^{n-1})$, dar distribuirea *DMUX*-ului în *DCD* și *MUX*-uri scade complexitatea sistemului pentru că

$$n + n2^n > (n-j)2^{n-j} + n-j + m(2^j + j2^j + j)$$

pentru un $m$ ce impune un *fan-out* rezonabil. Deci, pentru $j = \frac{n}{2}$,

$$S(n, m, j) = O(2^n),$$

față de $O(n2^n)$ pentru soluția din paragraful anterior.

Am *divizat* efortul de decodificare păstrînd complexitatea matricii programabile. "Divide et impera" este numele procedeului aplicat, prin care am distribuit decodificarea între o decodificare parțială, la intrare și o multiplexare, la ieșire. La acestea se mai adaugă și un alt efect pozitiv. Aria ocupată de matrice nu mai are laturile atît de disproporționate, la limită ar putea deveni chiar pătrată. Din punct de vedere tehnologic, uneori, acest efect este esențial.

### 3.3.4 Extinderea multiplexoarelor și demultiplexoarelor

Una din rațiunile ce stă la baza introducerii semnalului $E'$, care permite activarea unui *MUX*, *DMUX* sau unui alt circuit, este legată de extinderea capacității lor de lucru. Cu alte cuvinte, dispunînd de circuite de o anumită dimensiune să se poată construi un circuit cu aceeași funcție dar de dimensiuni mai mari.



• **Extinderea multiplexoarelor** este evidentă dacă se pornește de la *Definiția 3.4.* în care $f_i^j = W$, ieșirea *MUX*-ului cu $n$ intrări de selecție, iar $f_i^{j-n}$ sînt ieșirile a $2^n$ *MUX*-uri cu $j$-$n$ intrări de selecție. Se obține astfel un *MUX* cu $j$ intrări de selecție.

**Exemplul 3.29** Vom exemplifica pentru cazul în care $j = 5$ iar $n = 3$. Conform celor enunțate vom folosi un circuit cu 3 intrări de selecție (*MUX* de 8 căi) și 8 *MUX*-uri cu 2 intrări de selecție (*MUX*-uri de 4 căi). Interconectarea se face conform reprezentării din figura 3.34. Modul de funcționare este evident. Spre exemplu, dacă $S_4 S_3 ... S_0 = 11010$, atunci intrarea $I_6$ de la *MUX* este transferată la ieșire. Pe această intrare este aplicată ieșirea de la $MUX_6$ pe care se află valoarea de pe intrarea $I_2$ a acestuia. Deci la ieșirea *MUX*-ului extins se află valoarea de pe intrarea $I_{26}$. □

Extinderea se poate face și pe mai multe niveluri, intrările celui de al doilea nivel de *MUX*-uri putînd fi conectate la ieșirile altor circuite de același tip.

• **Extinderea capacității demultiplexoarelor** se face tot într-o structură arborescentă.

**Exemplul 3.30** Conform structurii din figura 3.35 unde intrarea $E'$ a *DMUX*-ului este împrăștiată către cele 8 *DMUX*-uri, $DMUX_i$, care, la rîndul lor, o mai demultiplexează o dată. Se obține un *DMUX* pe 32 de căi folosind unul de 8 căi și 8 de cîte 4 căi. Dacă $S_4 S_3 ... S_0 = 10111$, atunci ieșirea $O_5'$ a *DMUX*-ului se activează luînd valoarea lui $E'$ permițînd funcționarea circuitului $DMUX_5$ a cărui ieșire $O_3'$ este selectată de biții cei mai puțini semnificativi ai codului. Astfel intrarea $E'$ este transferată la ieșirea $O_{23}'$. □

Se pot imagina structuri pe mai multe niveluri ca și în cazul multiplexoarelor. Faptul că tranziția printr-un *DMUX* presupune, în circuitele concrete, parcurgerea

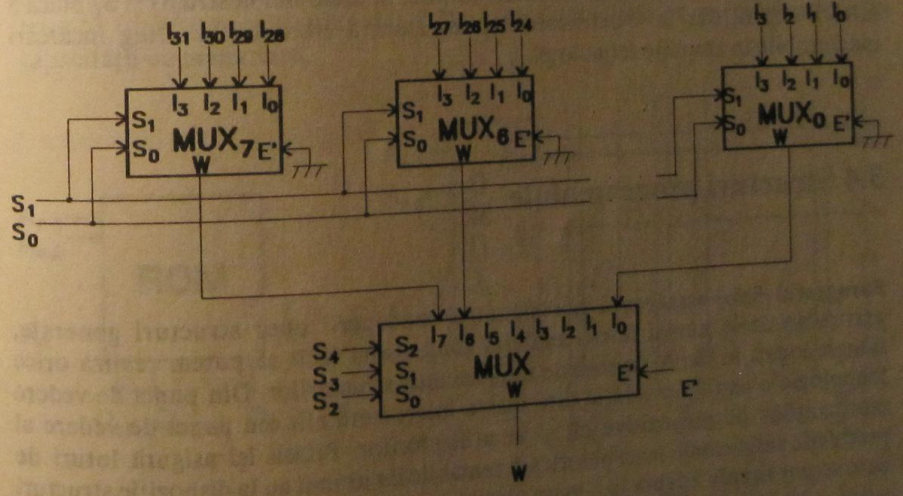



**Figura 3.34**

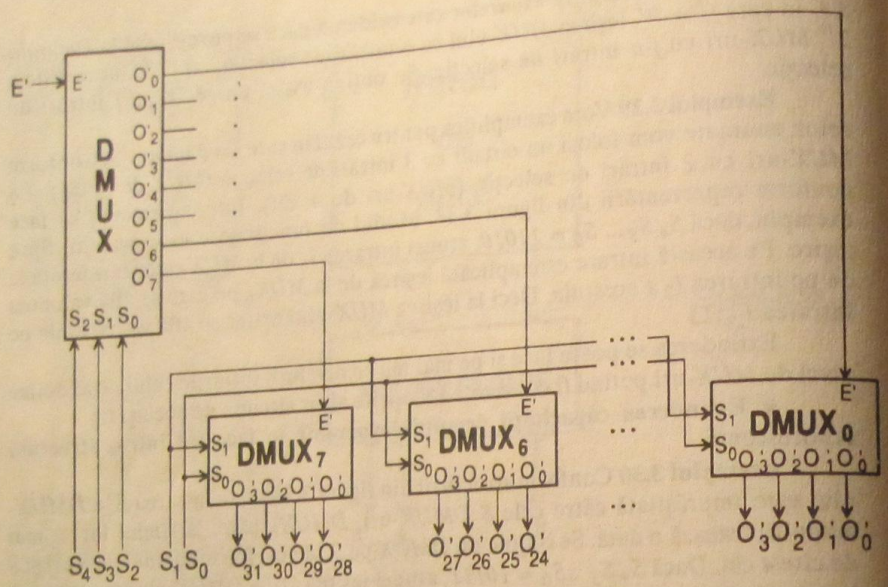



**Figura 3.35**

a două niveluri logice inversoare între $E'$ şi $O_i'$, asigură într-o structură extinsă deformarea şi întîrzierea minimă a semnalelor.

Atenţie la încărcarea (fan-out-ul) circuitelor ce generează semnalele de ordin inferior la intrarea unei structuri extinse. În exemplul nostru $S_1$ şi $S_0$ atacă *8* intrări numai, dar la o extindere suplimentară a arborelui se ating încărcări inadmisibile în anumite tehnologii.

## 3.4 Structuri programabile

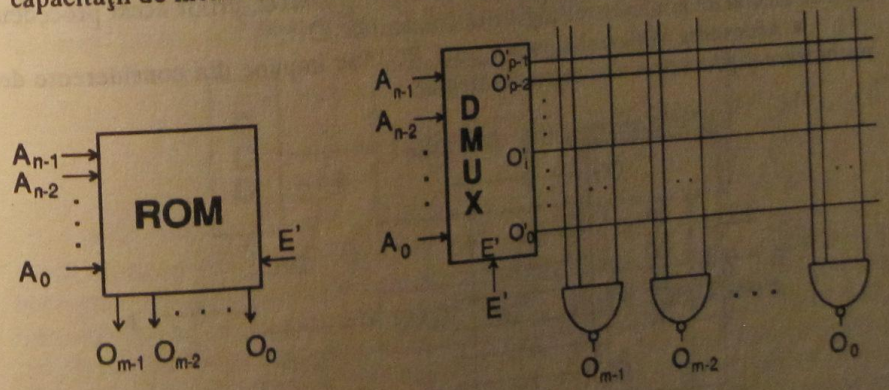
Paragraful 3.3.3 sugerează posibilitatea conceperii unor structuri generale, particularizabile numai într-o anumită zonă, astfel încît să putem realiza orice funcţie logică în limite impuse de dimensiunea structurilor. Din punct de vedere tehnologic o astfel de soluţie este foarte interesantă atît din punct de vedere al fabricanţilor de dispozitive cît şi al utilizatorilor. Primii îşi asigură loturi de producţie suficient de mari pentru a fi rentabile iar ultimii au la dispoziţie structuri care se pot simplu adapta la o mare diversitate de aplicaţii. Preţurile de cost pot să scadă astfel foarte mult facilitînd promovarea unei producţii foarte diverse în condiţii avantajoase.



### 3.4.1 Memorii fixe (*ROM*)

Subcapitolul anterior sugerează posibilitatea realizării unor structuri a căror funcţie să poată fi particularizată foarte simplu, prin realizarea unor conexiuni. Folosind o configuraţie de porţi logice *unică*, se poate obţine *orice* funcţie logică optînd ca, în puncte prestabilite, să se realizeze sau nu o conexiune electrică. Am văzut că realizarea unui *CLC* cu un *DMUX* şi o reţea de porţi se reduce la transpunerea tabelului de adevăr într-o configuraţie de conexiuni. Tabelul este transferat ca atare într-un ansamblu de conexiuni care reţine, *memorează*, configuraţia binară asociată unei funcţii logice. Pornind de la această observaţie se poate imagina un circuit logic universal cu structură fixă, particularizabilă prin modalitatea de interconectare dintre două nivele logice. Acest circuit va memora funcţia logică pe care utilizatorul o are în vedere.

Printr-un dispozitiv cu memorie înţelegem un circuit care poate regenera la comandă o configuraţie binară prestabilită. Dacă dorim să memorăm mai multe configuraţii binare ele vor fi ordonate şi astfel vor putea fi adresate, regăsite, prin numărul de ordine. Acest număr de ordine, care este tot o configuraţie binară, constituie adresa la care se regăseşte configuraţia memorată sau cuvîntul memorat, cum obişnuim să mai spunem. O memorie fixă stochează configuraţii binare care, odată înscrise (programate), nu mai pot fi modificate în funcţionarea normală a memoriei. Denumirea curentă este cea de *ROM* (read *o*nly *m*emory). Schema bloc a unui *ROM* este reprezentată în figura 3.36*a*. În acest *ROM* pot fi stocate $2^n$ configuraţii binare de cîte $m$ biţi fiecare. Adresarea se face prin codul $A_{n-1} A_{n-2} ... A_0$. Fiecărei adrese îi va corespunde o locaţie de memorie în care se află un cuvînt de $m$ biţi. Semnalul $E'$, cu aceeaşi semnificaţie ca cea de la circuitele anterior studiate, poate inactiva ieşirile ($E' = 1$) de data aceasta trecîndu-le în *Hi-Z*. Circuitele *ROM* au ieşirile cu colectorul liber (open collector) sau sînt de tip tristate (conform subcapitolului 9.2). Această facilitate va permite extinderea capacităţii de memorare.




a ) b )

**Figura 3.36**

În esenţă fiecărei configuraţii binare de adresare îi corespunde o configuraţie binară "stocată" în locaţia adresată. Cu alte cuvinte, funcţia de transfer a *ROM*-ului este o funcţie logică.

**Propoziţia 3.18** Un *ROM* cu $2^n$ locaţii de cîte $m$ biţi poate implementa *orice* funcţie logică cu $n$ intrări şi $m$ ieşiri. □

**Exemplul 3.31** Sumatorul complet de un bit poate fi implementat cu un *ROM* avînd $2^3 = 8$ adrese care ţintesc locaţii de doi biţi. Vom identifica adresele $A_2 A_1 A_0$ cu biţii de intrare *C, B* şi *A* iar cuvîntul de ieşire $O_1 O_0$ cu *S* şi $C_{out}$. Astfel la locaţia (adresa) *011* se află cuvîntul *01* indicînd faptul că $S = 0$ şi $C_{out} = 1$, adică: neavînd *C* (carry) pe intrare iar cei doi biţi de sumat avînd aceeaşi valoare, egală cu *1*, suma este *0* iar depăşirea *1*. □

• **Structura internă a unui ROM** ne este sugerată de modul de utilizare al *DMUX*-urilor la implementarea *CLC*-urilor. Structura potenţială, neactualizată pentru o funcţie oarecare, este cea din figura 3.36*b*. Ieşirile *DMUX*-ului *pot* fi interconectate cu oricare din intrările *NAND*-urilor astfel încît activarea ieşirii $O_i$' să poată determina comutatea oricărei ieşiri $O_j$ dacă se fac conexiunile corespunzătoare.

*ROM*-ul poate fi privit şi ca un *transcodificator* (transcodor), deoarece primul nivel *decodifică* configuraţia binară de intrare iar cel de al doilea nivel *codifică*. Într-adevăr, fiecărei intrări în nivelul de ieşire îi este asociată o configuraţie binară de ieşire. Prin modul de funcţionare al *DMUX*-ului este asigurată activarea a cel mult unei singure intrări. Un *ROM* realizează, astfel, o traducere; prin decodificare *recunoaşte* configuraţia binară iar prin codificare *generează* la ieşire configuraţia binară corespunzătoare.

**Exemplul 3.32** Reluînd *Exemplul 3.31* să concretizăm schema *ROM*-ului asociat, în figura 3.37 unde prin barele verticale am simbolizat simplificat cîte un *NAND* cu *8* intrări. Prin intersecţiile îngroşate am indicat care intrări au fost conectate (programate) la ieşirile *DMUX*-ului. Observăm faptul că proiectarea se reduce la transpunerea tabelului de adevăr al funcţiei în matricea de interconexiuni programabile. Ca şi în cazul folosirii *MUX*-urilor şi *DMUX*-urilor acest procedeu nu este afectat de minimizarea algebrică a funcţiilor. □

• **Adresarea bidimensională** a unui *ROM* se impune din considerente de optimizare şi geometrie a circuitului integrat.

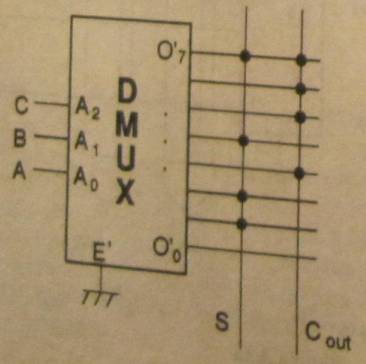



**Figura 3.37**



Dimensiunile uzuale ale acestor memorii fixe pornesc de la *32* de cuvinte de *8* biţi (*ROM* de *256* de biţi) şi ajung la capacităţi de ordinul megabiţilor (Mbit); sînt organizate pe cuvinte de *4* sau *8* biţi. Rezultă că prin creşterea capacităţii creşte şi *DMUX*-ul (decodificatorul) de adresare. La o memorie de *1* Mbit (organizată *256K×4*) acesta are *256×1024* (*256K*) ieşiri, deci ar trebui să fie realizat cu *256K* porţi cu *18* intrări fiecare. De asemenea matricea programabilă ar fi o structură practic filiformă de *256K* celule de *4* biţi fiecare. Realizarea geometriei circuitului integrat ar fi deosebit de dificilă. Adresarea bidimensională rezolvă ambele probleme.

Să considerăm varianta constructivă din figura 3.38, unde este reprezentat un *ROM* cu $2^n$ locaţii de $m = 4$ biţi. Soluţia este de tipul celei din figura 3.33. *DMUX*-ul în loc să selecteze un cuvînt de *4* biţi, va selecta unul de $4 \times 2^{n/2}$ biţi iar *MUX*-urile vor selecta în continuare din fiecare subcuvînt de $2^{n/2}$ biţi cîte un singur bit. Aria matricii programabile a rămas aceeaşi ca în caazul unui *DMUX* cu $n$ intrări, numai că raportul laturilor în loc să fie $2^n/4$ este *1/4*. Avantajul este indiscutabil.

Care sînt efectele asupra complexităţii circuitului? Vom aproxima, absolut acoperitor,

$$S_{DMUX}(n) = n2^n$$

$$S_{MUX}(n) = (n+1)2^n$$

şi vom considera, în ambele cazuri,

$$S_{Matrice\ programabilă} = m2^n.$$

În acest caz vom scrie pentru *ROM*-ul cu decodificator *(DMUX)* compact că avem

$$S_{ROM} = (m+n)2^n = O(n2^n)$$

iar pentru cel cu adresare bidimensională

$$S'_{ROM} = m2^n + n/2 \cdot 2^{n/2} + m(n/2+1)2^{n/2}.$$

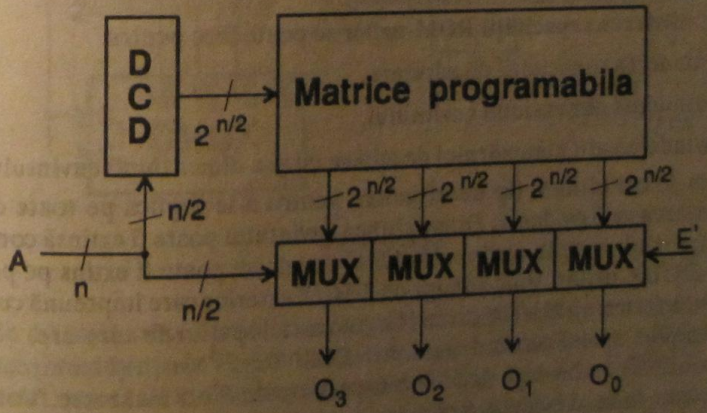



**Figura 3.38**

Pentru $m$ normal şi $n$ suficient de mare se poate scrie

$$S'_{ROM} \approx m2^n = O(2^n)$$

rezultînd că efectul decodificării distribuite bidimensional este acela că *circuitele de acces la matricea de memorie devin cantitativ nesemnificative.*

Putem, deci, susţine propoziţia ce urmează:

**Propoziţia 3.19** Complexitatea unui *ROM* adresat bidimensional cu adresă de $n$ biţi este $O(2^n)$ spre deosebire de *ROM*-ul adresat unidimensional (cu decodificator compact) pentru care complexitatea este $O(n2^n)$. □

**Exemplul 3.33** Pentru un *ROM* de $256K \times 4$ biti rezultă $n = 18$ şi $m = 4$. Vom calcula complexitatea în cele două cazuri şi vom obţine

$$S_{ROM} = 22 \cdot 2^{18}$$

$$S'_{ROM} = 4 \cdot 2^{18} + 49 \cdot 2^9 \approx 4 \cdot 2^{18}.$$

De peste cinci ori scade complexitatea memoriei fixe realizate cu adresare bidimensională. □

Cei $n$ biţi de adresă nu vor fi întotdeauna împărţiţi în două părţi egale. Uneori, în funcţie şi de valoarea lui $m$, o altă împărţire poate fi mai avantajoasă. Am considerat această situaţie numai pentru simplitatea calculului. Exemplul dat poate fi reluat cu *10* biţi către *DMUX* şi *9* catre multiplexoare, caz în care matricea programabilă devine pătrată. In esenţă efectul calitativ rămîne acelaşi. În funcţie de tehnologia folosită una sau alta dintre variante este acceptabilă depinzînd şi de raportul dimensiunilor celulei asociate fiecărui bit.

- Atunci cînd folosim într-o aplicaţie *ROM*-uri pentru implementarea *CLC*-urilor este foarte important să ţinem seama de faptul că dependenţa complexităţii de intrări este *exponenţială* iar de ieşiri numai proporţională. Cînd gîndim la nivel sistemic, la nivelul schemelor bloc, vom fi zgîrciţi în creşterea numărului de intrări a unui *CLC* şi ne vom permite o atitudine generoasă în definirea unui număr mare de ieşiri. Acest lucru este valabil pentru orice manieră de implementare, nu numai pentru cea care foloseşte memorii fixe. Efectul este însă cel mai evident în acest caz.

- **Extinderea capacităţii ROM-urilor** se poate face pentru

  - extinderea capacităţii de adresare,

  - extinderea dimensiunii cuvîntului,

  - extinderea atît a numărului de adrese cît şi a dimensiunii cuvîntului.

Vom folosi ultimul tip de extindere pentru a le explica pe toate deoarece particularizarea va fi evidentă. Dimensiunea cuvîntului poate fi extinsă configurînd alăturat cipuri adresate în comun. Numărul de adrese poate fi extins pe principiul extinderii *DMUX*-urilor. Vom adăuga un *DMUX* exterior care împreună cu cele din *ROM*-uri vor forma un *DMUX* extins, lărgind astfel spaţiul de adresare.

**Exemplul 3.34** Folosind memorii *ROM* de $2^n \times m$ biţi vom construi o memorie de $2^{(n+2)} \times 2m$ biţi, deci cu o capacitate de *8* ori mai mare. Vom utiliza pentru aceasta, în mod evident, *8* circuite conectate ca în figura 3.39. Conectarea în paralel a ieşirilor cipurilor de pe o verticală este posibilă datorită faptului că



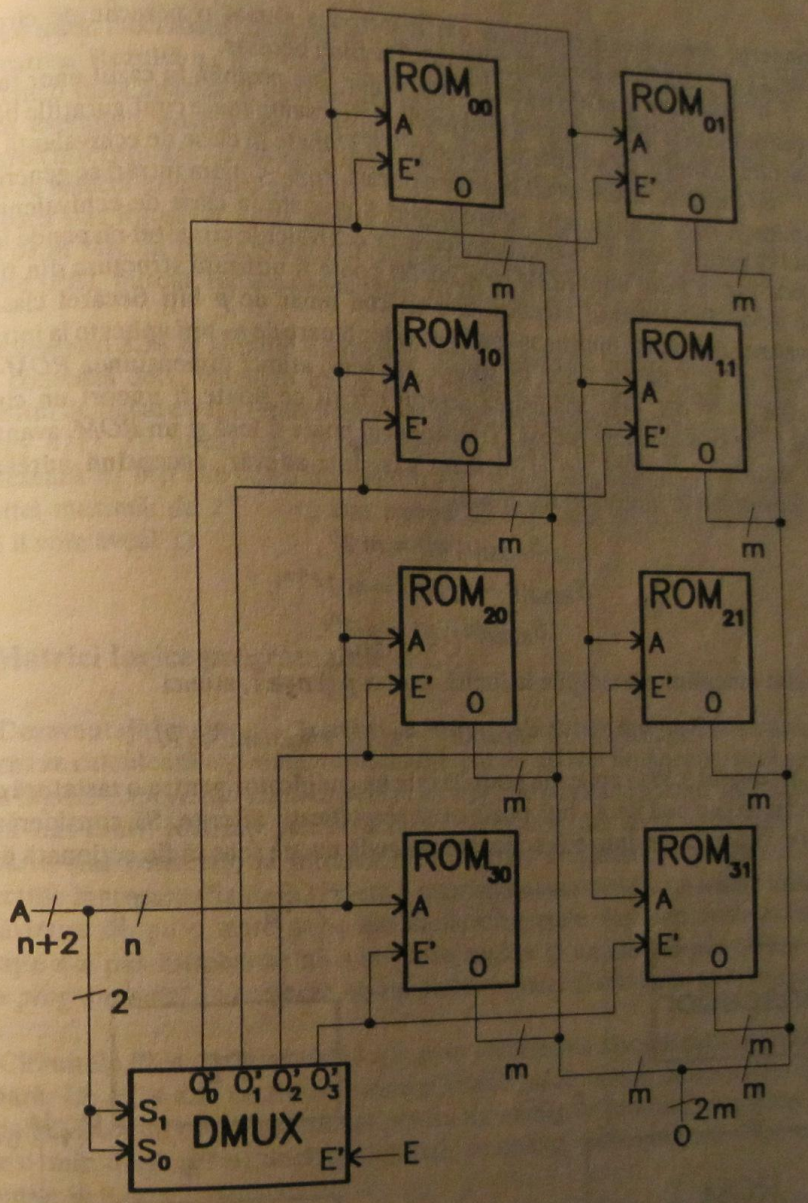



**Figura 3.39**

circuitele de ieşire sînt de tip tristate (dacă ar fi fost open collector ar fi trebuit conectate şi rezistenţe la $V_{CC}$ de la cele opt ieşiri ale sistemului). Deoarece la un moment dat numai cel mult o singură ieşire a *DMUX*-lui este activă $(O_i' = 0)$ numai o pereche, $ROM_{i0}$ şi $ROM_{i1}$, vor putea genera semnale de ieşire.

Dacă am fi dorit numai extinderea capacităţii de adresare schema ar fi conţinut numai circuitele $ROM_{i0}$ (sau $ROM_{i1}$), iar dacă s-ar fi dorit numai

extinderea dimensiunii cuvîntului am fi introdus numai o pereche de circuite, $ROM_{00}$ şi $ROM_{01}$ spre exemplu, *DMUX*-ul nefiind necesar.

• **Utilizarea mai eficientă a ROM-urilor** este posibilă în cazul unor funcţii parţial definite la intrare, atunci cînd nu sînt interesante toate configuraţiile binare de intrare în sensul că nu apar toate sau pot fi grupate în clase de echivalenţă.

Să considerăm un circuit logic cu $n$ intrări. Pe $n_1 < n$ din intrări se generează, în aplicaţia dată, configuraţii binare ce pot fi grupate în clase de echivalenţă, în sensul că pentru orice element al unei clase de echivalenţă circuitul răspunde la fel. În acest caz în locul unui *ROM* cu $n$ intrări poate fi utilizată structura din figura 3.40*a* unde *Transcodorul* asociază cîte un cod binar de $p$ biţi fiecărei clase de echivalenţă definită în mulţimea configuraţiilor binare de $n_1$ biţi aplicate la intrarea sa. Dacă $n_1 > p$, cînd o astfel de soluţie are sens, atunci dimensiunea *ROM*-ului scade de $2^{n_1-p}$ ori, prin adăugarea acestui circuit ce poate fi uneori un circuit standard de complexitate redusă. Transcodorul poate fi însă şi un *ROM*, avantajul fiind notabil de multe ori şi în acest caz. Într-adevăr, acceptînd adresarea bidimensională în interiorul *ROM*-ului,

$$S_{ROM}(n, m) = m\,2^n,$$
$$S_{ROM}(p + n_2, m) = m\,2^{p+n_2},$$
$$S_{ROM}(n_1, p) = p\,2^{n_1}.$$

Dacă sînt indeplinite condiţiile la limită ca $m \geq p$ şi $n_2 \geq 1$, atunci

$$S_{ROM}(n, m) \geq S_{ROM}(p + n_2, m) + S_{ROM}(n_1, p).$$

**Exemplul 3.35** O aplicaţie posibilă este un codificator pentru o tastatură care în funcţie de un cod de $n_2$ biţi poate avea semnificaţii diferite. Să considerăm o claviatură cu $n_1$ taste. Într-o funcţionare normală nu are sens să fie acţionată decît

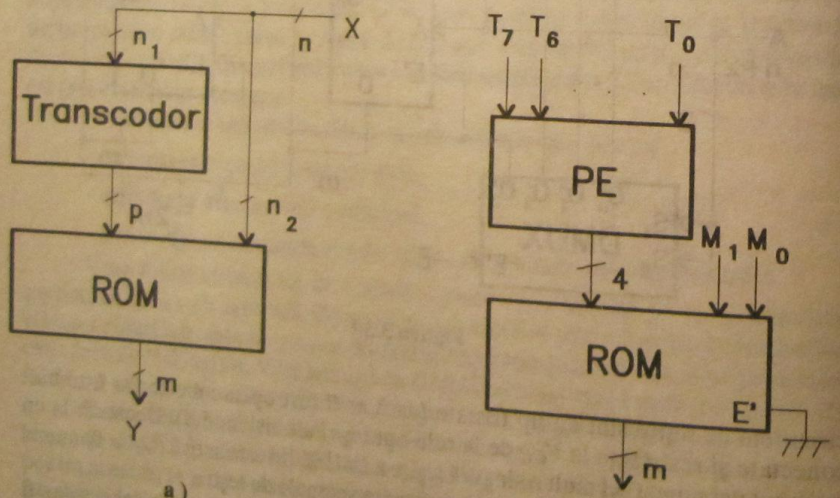






**Figura 3.40**



cel mult o tastă. Deci toate codurile care au mai mult de un bit cu valoarea *1* nu sînt semnificative. Rezultă $n_1 + 1$ clase de echivalenţă în cele $2^{n_1}$ coduri binare ($n_1$ pentru cîte o tastă apăsată şi unul pentru cazul cînd niciuna nu este activată).

În exemplul nostru vom considera *8* taste şi patru moduri distincte de codificare (interpretare) a tastelor, deci $n_1 = 8$ şi $n_2 = 2$ (prin două comutatoare putîndu-se selecta cale patru moduri).

Dacă s-ar folosi un *ROM* pentru această codificare ar fi necesar unul de *1K* cuvinte de $m$ biţi. În figura 3.40*b* se observă că prin folosirea unui codificator cu prioritate *(PE)* putem reduce capacitatea *ROM*-ului la *64* de locaţii (deci de *16* ori). □

**Exemplul 3.36** O situaţie limită este aceea în care de fiecare dată din cei $n_1$ biţi nu contează decît valoarea unuia singur pentru a determina configuraţia de ieşire. Bitul de fiecare dată semnificativ este indicat de un cod suplimentar de $q$ biţi ($q \geq \lceil \log_2 n_1 \rceil$). În acest caz *Transcodorul* se poate reduce la un *MUX* ce multiplexează $n_1$ biţi sub comanda celor $q$ biţi. Reducerea capacităţii *ROM*-ului este astfel maximă: de $2^{n_1-1}$ ori. Dar trebuie să dispunem de codul de selecţie. Uneori îl vom avea! □

## 3.4.2 Matrici logice programabile *(PLA)*

Dezavantajul major al soluţiei cu *ROM*-uri este dat de faptul că în structura sa internă se calculează *toţi mintermenii* indiferent de cîţi sînt utili pentru realizarea funcţiei logice. Experienţa arată că pentru o categorie largă de aplicaţii majoritatea acestora sînt inutil realizaţi prin structura *DMUX*-ului, ale cărui ieşiri numai în mică parte sînt conectate la intrarea *NAND*-urilor din matricea programabilă. Intersecţiile îngroşate (figura 3.37) într-o matrice sînt deseori foarte rare şi lipsesc cu desăvîrşire de pe o mare parte din liniile orizontale. Pornind de la această observaţie s-a pus întrebarea: nu s-ar putea realiza şi un *DMUX parţial* dar cu *produse programabile?* În acest caz ne-am putea permite şi creşterea numărului de intrări.

Circuitele *PLA* (*p*rogramable *l*ogic *a*rray) reprezintă răspunsul la întrebarea anterioară. *Un PLA este un ROM la care şi DMUX-ul de intrare este programabil.* Potenţial, *DMUX*-ul este dimensionat pentru un număr de produse (mintermeni) mult mai mic decît cel al unei configuraţii complete. Schematic un *PLA* este reprezentat în figura 3.41 unde:

- liniile verticale îngroşate reprezintă *48* de *NAND*-uri, cu *32* de intrări fiecare conectabile la $I_i$ sau $I_i'$ $(i = 0, ..., 15)$, cîte unul pentru cele *48* de produse elementare $P_j$ $(j = 0, ..., 47)$ ce pot fi definite la acest nivel;

- liniile orizontale îngroşate reprezintă *8* *NAND*-uri cu *48* de intrări conectabile la ieşirile porţilor $P_j$;

- *XOR*-urile de pe cele *8* ieşiri permit negarea programată a ieşirilor celor *8* *NAND*-uri (programarea se face întrerupînd, prin ardere, rezistenţele conectate la

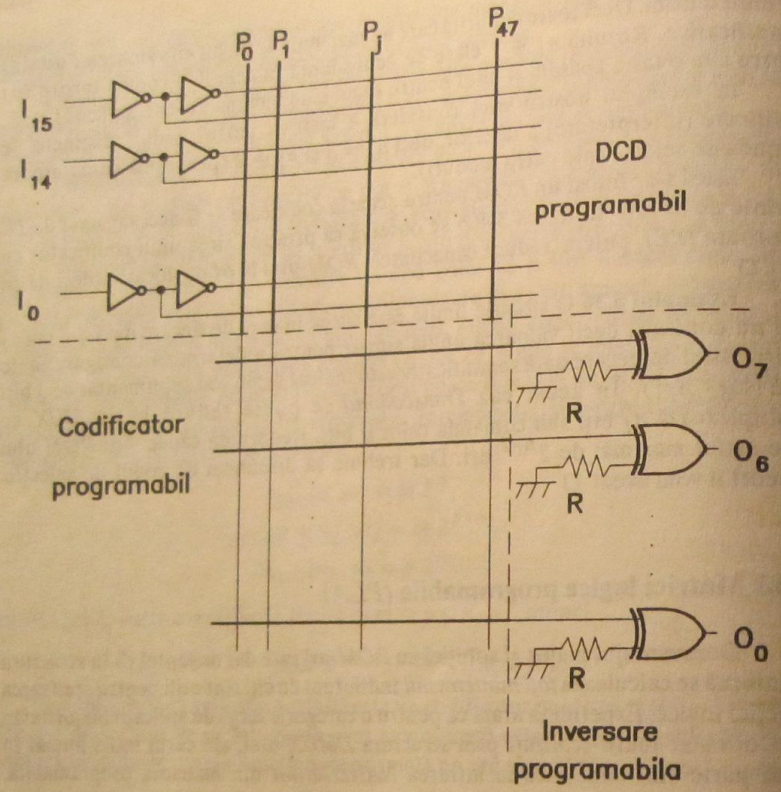



**Figura 3.41**

masă; se lasă în gol intrarea, ceea ce echivalează cu *1* aplicat intrării prin care se comandă negarea).

Un număr foarte mare de *CLC*-uri cu $n=16$ şi $m=8$ vor putea fi implementate cu acest *PLA*. Nu toate însă. Şansele ne sînt sporite de posibilitatea de a nega, oferită de *XOR*-uri, de multe ori funcţia negată presupunînd un număr mult mai mic de produse. Limitarea numărului de produse, impusă de această structură constituie restricţia cea mai importantă. Deci funcţiile logice la care *1*-urile din tabel sînt sensibil apropiate ca număr de cel al *0*-urilor, nu vor putea fi implementate cu un astfel de procedeu.

Este evident că un *XOR* cu *16* intrări nu poate fi implementat pentru că necesită $2^{15}$ produse elementare (*Propoziţia 3.11*). Această observaţie ne atrage atenţia asupra faptului că *implementarea funcţiilor de tip aritmetic nu-şi va găsi o soluţie eficientă pe această cale.*

*Folosirea PLA-urilor impune procedee de minimizare* deoarece este o structură cu resurse limitate (numărul produselor).



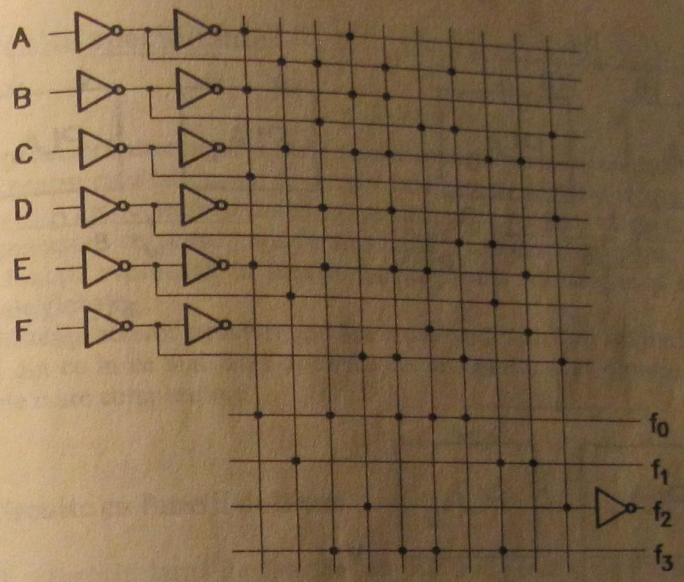



**Figura 3.42**

**Exemplul 3.37** *PLA*-ul din figura 3.42 are următoarele funcţii de ieşire:

$$f_0 = ABC'E + A'B'DE + A'BCDEF' + B'EF + A'B'D'F',$$

$$f_1 = A'CE' + CD'E' + CEF,$$

$$f_2 = (ABCF' + B'DF')',$$

$$f_3 = A'B'DE + A'BCDEF' + B'EF + CD'E'.$$

Negarea pentru $f_2$ a fost obţinută distrugînd, printr-un curent suficient de mare, rezistenţa de la intrare *XOR*-ului. Indicaţii pentru programarea *PLA*-ului se găsesc în cataloagele producătorilor. □

Observăm faptul că *un produs programabil poate fi asociat mai multor mintermeni dacă aceştia sînt adiacenţi.* În exemplul anterior produsul *ABC'E*, spre exemplu, corespunde la patru mintermeni.

- Atunci cînd se proiectează structura unui circuit integrat se folosesc pentru *CLC*-uri configuraţii de tip *PLA*. Structura de *ROM* este lipsită de sens în acest caz. *ROM*-ul poate minimiza numărul de cipuri utilizate în realizarea unui sistem. *PLA*-ul minimizează aria pe care se realizează un circuit integrat.
- În cazul *CLC*-urilor *cu funcţie nenumerică*, pentru un număr de intrări mai mare de *10÷12* sînt recomandabile circuite de tip *PLA*. Folosirea *ROM*-urilor la astfel de dimensiuni începe să fie ineficientă.
- **Extinderea PLA-urilor** se referă la

- creşterea numărului de produse (figura 3.43*a*),

- creşterea numărului de intrări (figura 3.43*b*),

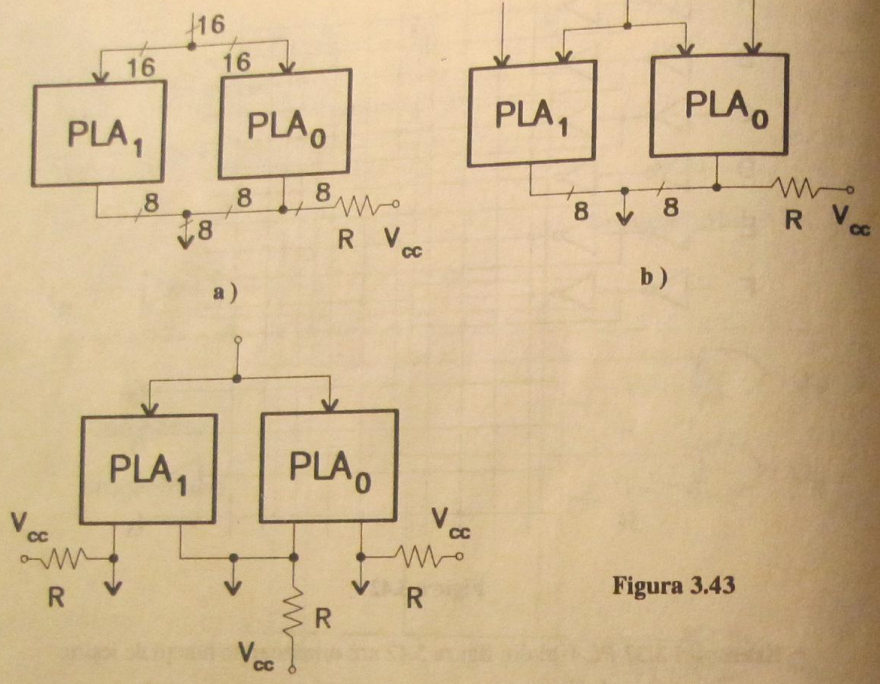



Figura 3.43

- creşterea numărului de ieşiri (figura 3.43*c*).

Circuitele folosite în extinderile anterioare au ieşirile de tip open- collector.

• **Matricile de porţi logice - PGA** (*p*rogramable *g*ates *a*rray) reprezintă nişte decodificatoare programabile; în esenţă fiind vorba de primul nivel al unui *PLA*. În familiile standard de circuite se găsesc *PGA*-uri cu *16* intrări şi tot atîtea ieşiri cu ajutorul cărora se pot calcula oricare *16* mintermeni din cei *65536* posibili.

• **Ariile de porţi logice** (gates arrays) reprezintă o tehnologie modernă de realizare la comandă a unor circuite foarte complexe în condiţii de foarte mare diversitate. Într-un regim *semicustom design* la fabricant se procesează plachete cu cipuri pe care se află matrici de porţi neconectate între ele sau parţial conectate în configuraţii standard. La cererea clientului se realizează măştile, prin care se definesc interconexiuni ce corespund unei aplicaţii concrete şi se încheie procesarea plachetelor realizîndu-se loturi de circuite la cererea clientului. Din punct de vedere teoretic această abordare nu pune probleme deosebite, motiv pentru care nu vom insista. Practic reprezintă însă un procedeu tehnologic foarte folosit şi de mare perspectivă datorită instrumentelor automate de proiectare care permit o actualizare foarte rapidă şi eficientă a celor mai diverse funcţiuni.



## 3.5 Circuite specializate

Practica proiectării sistemelor digitale a pus în evidenţă anumite funcţii de utilitate generală pentru care fabricanţii de circuite integrate au definit şi produs circuite specializate. A apărut astfel o clasă largă de circuite cu funcţiuni uzuale şi facilităţi de interconectare ce permit structurarea facilă a sistemelor complexe. Spectrul acestor funcţii evoluează în funcţie de progresele tehnologice şi cerinţele funcţionale globale.

Existenţa acestor structuri rămîne importantă chiar în condiţiile integrării pe scară din ce în ce mai largă deoarece ele se regăsesc ca subsisteme ale unor sisteme de mare complexitate.

### 3.5.1 Circuite cu funcţii dedicate

O categorie largă de funcţii este cea care permite soluţionarea unor probleme numerice. Ele s-au conturat mai bine datorită caracterului preponderent numeric pe care-l au încă aplicaţiile circuitelor logice.

• **Sumatorul binar** este o structură ce se găseşte curent în proiectarea sistemelor digitale. O structură standard este sumatorul de patru biţi ( figura 3.44) ce realizează suma a două cuvinte de *4* biţi, $A_3A_2A_1A_0$ şi $B_3B_2B_1B_0$, cu $C_{in}$ (carry de la ordinul binar anterior), generînd şi $C_{out}$ (depăşire, transport, pentru ordinul binar următor). Caracteristic pentru acest circuit este că evaluează pe $C_{out}$ numai pe două niveluri logice, asigurînd o propagare rapidă a transportului în cazul în care sînt conectate *n* astfel de sumatoare pentru a forma unul de *4n* biţi. Conectarea se face după regula sugerată în figura 3.22*b*. Pentru *n* suficient de mare limitarea de viteză a circuitului este dată de propagarea transportului. Pentru detalii se pot consulta cataloagele de firmă la codul *7483* (*CDB 483*).

• **Comparatorul** de patru biţi este tot un circuit cu o coloratură aritmetică. Pe cele trei ieşiri, $A=B$, $A<B$, $A>B$, este calculată relaţia în care se află cele două numere, reprezentate pe patru biţi, aplicate intrărilor. Circuitul posedă şi trei intrări, notate precum cele trei ieşiri, destinate extinderii capacităţii funcţionale la compararea cuvintelor de *4n* biţi. În figura 3.45 este reprezentat un comparator de *8* biţi realizat cu două circuite standard de tip *7485*. Conexiunea se face, în acest caz,

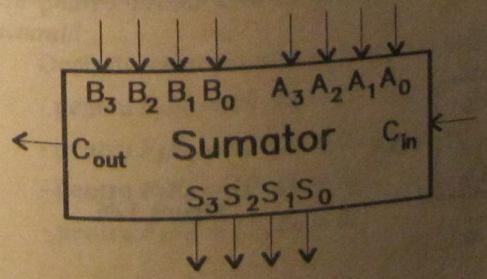



Figura 3.44

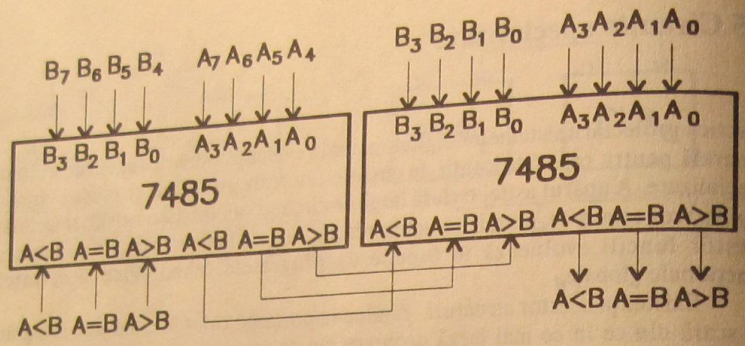



Figura 3.45

de la ordinele binare superioare către cele inferioare, spre deosebire de sumator la care semnalul de depăşire merge către ordinele binare mari.

• **Codificatorul cu prioritate** a fost definit în *Exemplul 3.15*. Circuitul se regăseşte în seriile standard sub codul *74148*. Notăm faptul că intrările şi ieşirile sînt active pe *zero*. Pentru extinderea funcţionării pot fi luate în consideraţie două soluţii, una care creşte minimal complexitatea sistemului şi alta care minimizează creşterea timpului de propagare. Cititorul este invitat să le conceapă.

Interpretarea numerică a funcţiei acestui circuit o reamintim: calculează partea întreagă a logaritmului în bază doi din numărul de opt biţi aplicat la intrare.

### 3.5.2 Circuite cu funcţii selectabile

Aplicaţiile curente impun ca fluxul de date într-o porţiune a unei maşini să fie prelucrat diferit în momente de timp distincte. Apare astfel necesitatea unor circuite care să aibă o funcţie de transfer comandată.

• Un prim exemplu îl constituie circuitul care are structura de principiu dată în figura 3.46, unde:

- $I_3 ... I_0$ sînt intrările de date;

- $O_3 ... O_0$ sînt ieşirile de date;

- $F_1F_0$ permit aplicarea *codului de funcţie* astfel încît pe ieşiri să avem $O_i = 0$ sau $O_i = 1$ sau $O_i = I_i$ sau $O_i = I_i'$ pentru $i = 0, ...,3$, în funcţie de valoarea concretă aplicată pe aceste două intrări;

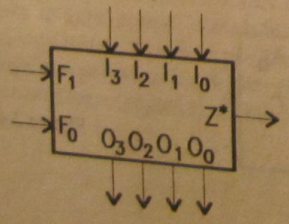



Figura 3.46



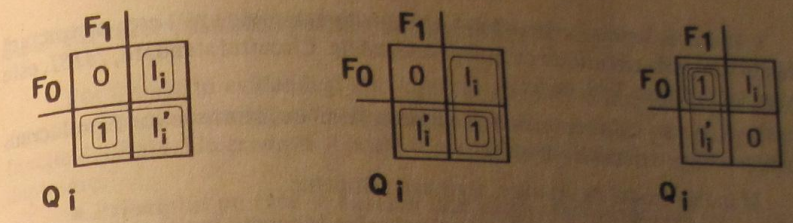



Figura 3.47

- $Z^*$ este o ieşire care ne indică faptul că pe intrările date toţi biţii sînt egali cu zero ($Z = I_3'I_2'I_1'I_0'$); *această ieşire este de tip open-collector* pentru a putea fi extinsă, prin conectare în paralel cu altele identice, la configuraţii de *4n* biţi.

Cum alocăm cele patru comportamente ale ieşirii celor patru coduri de comandă? În figura 3.47 sînt prezentate trei variante semnificative. Analizînd dimensiunile suprafeţelor definibile, variantele *b* şi *c* sînt cele mai eficiente. Cum alegem între ele? Extrăgînd funcţiile:

$$O_i = F_1'F_0' + F_0'I_i' + F_1'I_i$$
$$O_i = F_1F_0 + F_0I_i + F_1I_i'$$

observăm că prima presupune variabilele binare $F_1$ şi $F_0$ negate iar cea de a doua variabilele nenegate. Pentru $i = 0, ...,3$ vom folosi intrările $F_1$ şi $F_0$ de patru ori. Pentru a mări *fan-out*-ul circuitelor ce comandă aceste intrări, se impun inversoare pentru aceste variabile (ca şi în cazul *MUX*-urilor sau *DMUX*-urilor). Dacă vom opta pentru prima variantă, (*b*), atunci vom folosi numai un rînd de inversoare pe intrare, pentru că în definirea funcţiei nu intră decît $F_1'$ şi $F_0'$. Deci funcţia de transfer va fi

$$O_i = ((F_1'F_0)'(F_0'I_i')'(F_1I_i)')'.$$

Produsul $(F_1'F_0')'$ va fi calculat numai o singură dată.

Observăm că nu este indiferentă codificarea funcţiilor. Pentru o anumită codificare structura obţinută este minimală ca adîncime şi complexitate.

*Codul de selecţie poate fi stabilit arbitrar.* Această *libertate în codificare*, ce apare pentru prima dată în acest curs, este şi *va fi folosită pentru optimizare structurală*.

Deci, structura noastră va funcţiona astfel:

- pentru $F_1F_0 = 00$, $O_i = 1$,
- pentru $F_1F_0 = 01$, $O_i = I_i'$,
- pentru $F_1F_0 = 10$, $O_i = I_i$,
- pentru $F_1F_0 = 11$, $O_i = 0$.

• **Unitatea logico - aritmetică** - ALU (*a*ritmetic *l*ogic *u*nit) este o structură tipică din categoria circuitelor cu funcţii selectabile. Circuitul standard, *74181*, este reprezentat în figura 3.48, unde:

- $A_3...A_0$, $B_3...B_0$ reprezintă cele două mărimi de intrare ce vor fi prelucrate logic ($M = 0$) sau aritmetic ($M = 1$);

- $M$ indică modul de operare, logic sau aritmetic;

- $S_3...S_0$ reprezintă codul de selecţie al celor *16* funcţii logice sau al celor *16* funcţii aritmetice;

- $F_3 ...F_0$ constituie ieşirea pe care apare rezultatul operaţiilor logice sau aritmetice;

- $C_{in}$ şi $C_{out}$ au semnificaţia de la sumatorul de patru biţi dar numai în cazul în care sînt comandate funcţii aritmetice (atunci cînd se face scădere au semnificaţia de împrumut (borrow));

- $P$ şi $G$ sînt semnale care permit cu ajutorul unui alt circuit *(74182)* calculul rapid al transportului (împrumut sau depăşire);

- $A = B^*$ este o ieşire open-colector care indică faptul că cei doi operanzi de pe intrare sînt identici (se poate expanda prin simplă conectare în paralel cu alte ieşiri de acelaşi tip).

Pentru detalii se poate consulta orice catalog de circuite integrate digitale, în care se vor găsi listate cele *32* de funcţii ale *ALU*.

• **Categoriile de intrări şi ieşiri** ce pot apare la un astfel de circuit sînt următoarele:

- intrări de date ($A$ şi $B$);

- ieşiri de date ($F$);

- intrări de comandă ($M$ şi $S$);

- intrări ce caracterizează funcţia comandată ($C_{in}$, adunare cu sau fără transport de la ordinul binar anterior);

- ieşiri ce asigură extensia ($C_{out}$, $P$, $G$, $A=B^*$);

- ieşiri ce permit caracterizarea rezultatului operaţiei efectuate ($C_{out}$, $A=B^*$), putînd fi utilizaţi ca indicatori.

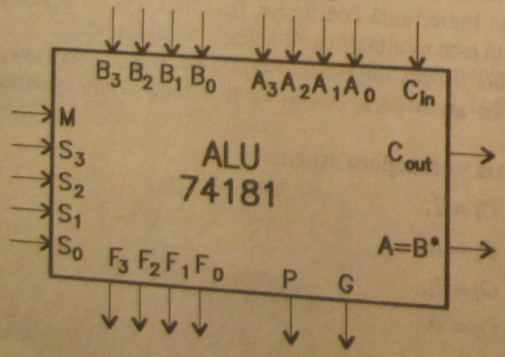



Figura 3.48



## 3.5.3 Circuite combinaţionale de deplasare

Pot fi puse în evidentă şi alte tipuri de circuite, la care pe lîngă *intrarea de date* să existe o comandă pe care se *impune cantitativ* modul în care se efectuează o funcţie dedicată. Un exemplu îl poate constitui clasa circuitelor combinaţionale de deplasare.

• **Principiul** pe care se bazează deplasarea combinaţională este ilustrat în figura 3.49, unde este reprezentată o porţiune dintr-un astfel de circuit. Notăm cu $D$ numărul format din $D_1D_0$ rezultînd că

$$O_i = I_{i+D}.$$

Prin modul în care a fost realizată conectarea intrărilor *MUX*-urilor, codul de selecţie determină regăsirea configuraţiei binare $...I_n I_{n-1}...$ nedeplasată ($D_1D_0 = 00$), sau deplasată cu una, două sau trei poziţii la stînga. O altă interpretare posibilă este cea a înmulţirii cu puteri ale lui doi. Într-adevăr, dacă notăm cu $I$ configuraţia binară de intrare, cu $O$ pe cea de ieşire, putem scrie că

$$O = I \cdot 2^D.$$

Orice *shift*-are la stînga presupune înmulţirea cu o putere a lui doi, iar o shift-are la dreapta este echivalentă cu împărţirea la o putere a lui doi.

Codul $D$ comandă cantitativ unica funcţie pe care o are circuitul.

Pe acelaşi principiu se poate realiza un circuit de shift-are la dreapta. Cititorul poate desena, ca exerciţiu, acest circuit.

• **Rotorul** (barrel shifter) este realizat pe acelaşi principiu determinînd nu numai o simplă deplasare ci o rotire a configuraţiei binare. Prin rotire, la stînga spre exemplu, biţii ce depăşesc la stînga capacitatea de reprezentare sînt introduşi pe poziţiile cele mai puţin semnificative. Spre exemplu, configuraţia binară *10001010* rotită cu o poziţie la stînga va deveni *00010101*, cel mai semnificativ *1* regăsindu-se pe prima poziţie (cea mai puţin semnificativă).

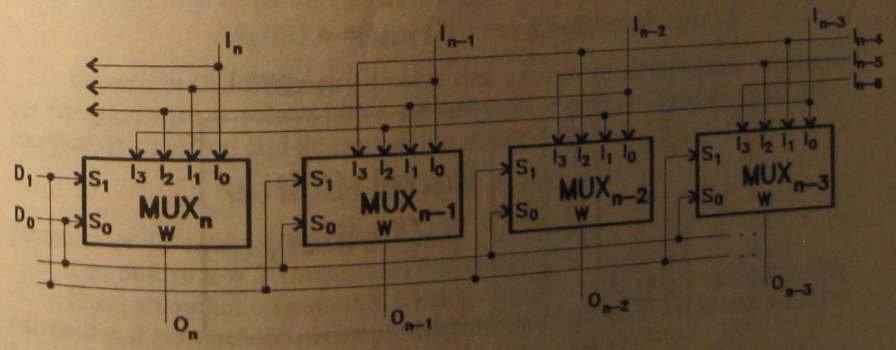



Figura 3.49

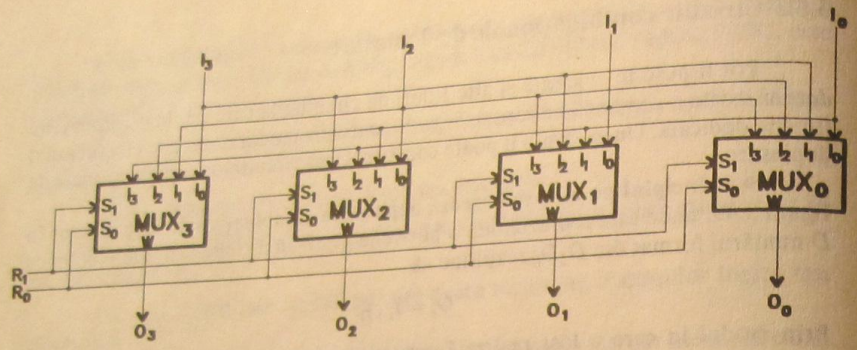



Figura 3.50

Pentru exemplificare am reprezentat în figura 3.50 un rotitor pentru cuvinte de *4* biţi.

Rotirea la dreapta impune o altfel de structură? Nu. Este suficient că complementăm faţă de doi codul $R_1R_0$ şi obţinem o rotire la dreapta. Într- adevăr, rotirea la stînga cu o poziţie este echivalentă cu rotirea la dreapta cu trei poziţii, în exemplul anterior.

Rezultă că pentru rotire bidirecţională putem folosi schema din figura 3.51 în care circuitele *XOR* şi *INC* complementează faţă de 2, comandat de codul *R*. Pentru $S = 0$ se realizează rotire la stînga iar pentru $S = 1$ rotire la dreapta.

• **Complexitatea şi adîncimea rotorului** variază în funcţie de implementare. În varianta prezentată

$$D_R(n) = 2\ ,$$

$$S_R(n) = nS_{MUX}(n) = n^2(2 + \log_2 n)$$

dacă notăm cu $n$ numărul de biţi ai configuraţiei binare rotite.

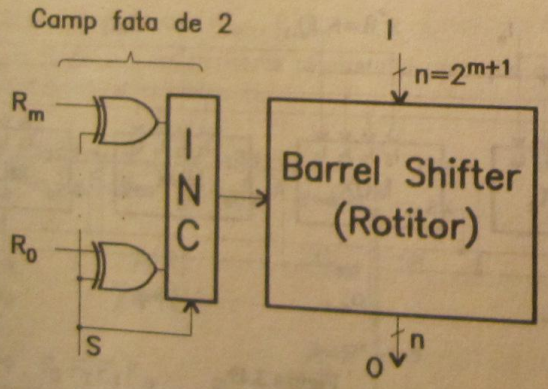



Figura 3.51



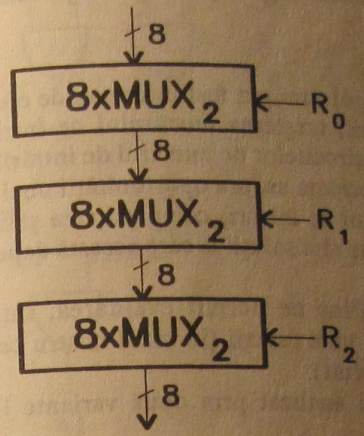



Figura 3.52

O altă variantă de implementare presupune realizarea a $\log_2 n$ shift-ere fiecare realizînd o rotire cu cîte o putere a lui doi începînd cu *1* şi încheind cu $n/2$. În figura 3.52 este reprezentat un astfel de rotitor pentru $n = 8$. La fiecare nivel de *8 MUX*-uri de cîte două căi fiecare, configuraţia binară poate trece nerotită sau rotită cu o poziţie ($R_0$), cu două poziţii ($R_1$) sau cu patru poziţii ($R_2$). Dacă $R_2R_1R_0 = 101$, atunci primul nivel roteşte cu o poziţie ($R_0 = 1$), următorul nu roteşte, lăsînd să treacă neschimbată configuraţia binară ($R_1 = 0$), iar ultimul roteşte cu patru poziţii ($R_2 = 1$). Rezultă în final o rotire cu *5* poziţii (*101*). Adîncimea unui rotor realizat după acest principiu este

$$D_{R'}(n) = D_{MUX}(2) \cdot \log_2 n = 2\log_2 n$$

Complexitatea va fi

$$S_{R'}(n) = nS_{MUX}(2) \cdot \log_2 n = 6n \log_2 n$$

Observăm că adîncimea a crescut, deci a scăzut viteza de lucru a circuitului, dar complexitatea a scăzut într-o măsură mult mai mare. Produsul adîncime - complexitate a scăzut faţă de cazul anterior. Într- adevăr:

$$D_R(n) \cdot S_R(n) \geq D_{R'}(n)S_{R'}(n)$$

pentru că $2n^2(2 + \log_2 n) \geq 12n(\log_2 n)^2$.

Dar pentru $n > 2$ inegalitatea este strictă.

Dacă acceptăm o anumită scădere a vitezei de lucru atunci sîntem răsplătiţi cu o mult mai mare scădere a complexităţii sistemului. Un rezultat identic a fost pus în evidenţă în cazul *Exemplului 3.22*.

• Cum poate fi folosit un rotitor pentru realizarea deplasării la stînga sau la dreapta? Propunem această problemă cititorului.

## 3.6 Complexitatea circuitelor logice combinaţionale

De mai multe ori, pe parcursul acestui curs, am fost interesaţi de efectul pe care-l are, asupra dimensiunii unui circuit, creşterea numărului de intrări. Pentru o funcţie dată, cum depinde mărimea circuitelor de numărul de intrări? Clarificarea acestei dependenţe ne permite să decidem asupra oportunităţii unei soluţii. Dacă odată cu creşterea lui $n$ (a numărului de intrări) complexitatea şi/sau adîncimea unui circuit cresc prea mult vom căuta alte soluţii la care această dependenţă să fie mai convenabilă.

• **Notaţiile pentru magnitudine** ne permit evaluarea, într-o manieră esenţializantă, a dependenţei de $n$ a unei funcţii (în cazul nostru cea care ne dă complexitatea sau adîncimea unui circuit).

**Exemplul 3.38** Pentru rotorul analizat prin două variante în paragraful anterior se poate scrie

$$D_R(n) = O(1),$$
$$S_R(n) = O(n^2 \log n);$$
$$D_{R'}(n) = O(\log n),$$
$$S_{R'}(n) = O(n \log n).$$

Specificarea bazei logaritmului devine inutilă datorită faptului că $c$ (Capitolul 9.1) poate lua orice valoare. □

Această notaţie este mai sugestivă, redînd dependenţa esenţială de valoarea lui $n$.

• **Formulă - circuit** reprezintă o alternativă teoretică deosebit de semnificativă în teoria sistemelor logice.

**Definiţia 3.7** O reţea de porţi logice reprezintă o *formulă* dacă fan-out-ul oricărui circuit este egal cu unu. □

**Definiţia 3.8** O reţea de porţi logice reprezintă un *circuit* dacă există porţi cu fan-out-ul mai mare decît unu. □

O variantă echivalentă a definiţiilor anterioare este următoarea:

**Definiţia 3.9** O reţea de porţi reprezintă o *formulă* dacă are asociată o structură de tip arbore sau un *circuit* dacă are asociat o structură de tip graf. □

**Exemplul 3.39** *CLC*-ul din figura 3.53*a* este o *formulă* pe cînd cel din figura 3.53*b* este un *circuit* pentru că unul din *NAND-uri* are *fan-out* = 2. Un *circuit* poate fi transformat în *formulă* multiplicînd porţile cu *fan-out* >1. În figura 3.53*c* este reprezentată *formula* asociată *circuitului* din figura 3.53*b*. □

**Propoziţia 3.20** Dacă notăm cu $S_F(n)$ complexitatea unei *formule* şi cu $S_C(n)$ pe cea a unui *circuit*, atunci pentru acelaşi *CLC* putem scrie:

$$S_F(n) > S_C(n). \square$$

Termenii de *formulă* şi *circuit* provin de la faptul că o subexpresie este scrisă de cîte ori apare în expresia unei funcţii logice, pe cînd într-un circuit o poartă, sau



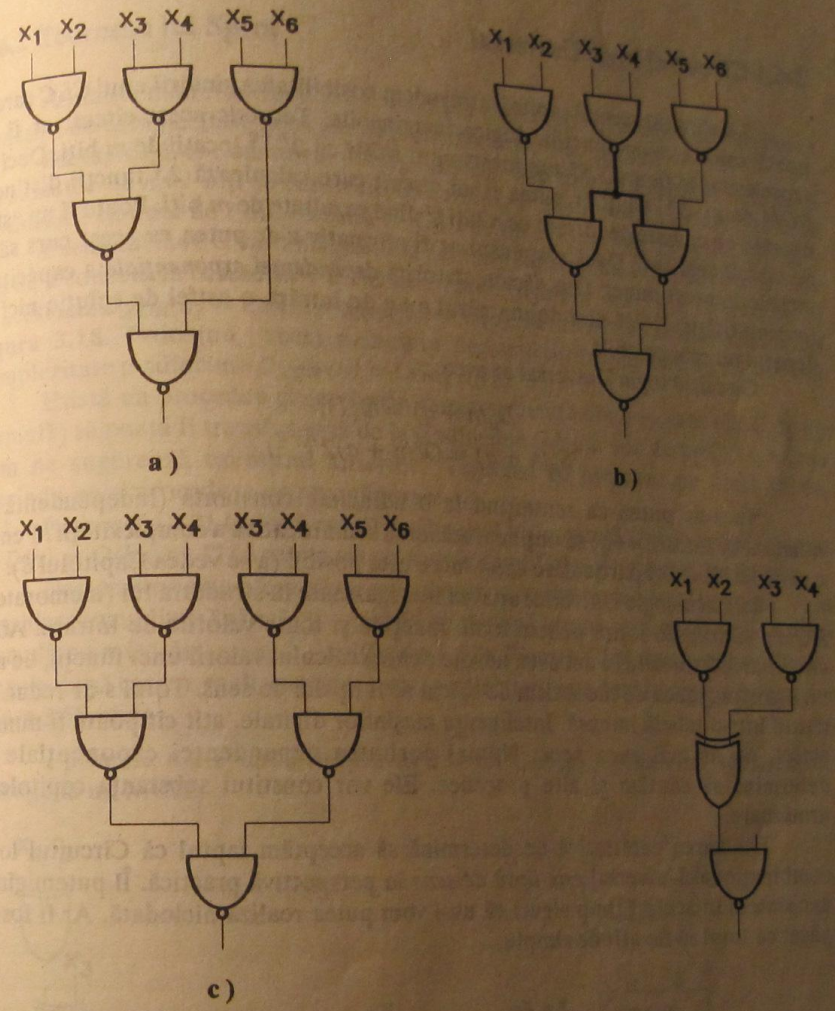



**Figura 3.53**




**Figura 3.54**

o structură oarecare, pot fi realizate o singură dată şi conectate la intrarea mai multor porţi.

Să considerăm *CLC*-ul din figura 3.54. Reprezintă o *formulă* sau un *circuit*? Aşa cum este reprezentat este o *formulă*. Dar dacă se detaliază circuitul *XOR*, atunci va deveni un *circuit*. Deci caracterul de *formulă* sau *circuit* va fi dat de funcţiile elementare luate în consideraţie. Din acest punct de vedere structura din figura 3.22*b* este o *formulă* pentru că substructurile elementare luate în consideraţie sînt sumatoarele complete de un bit. Dacă acestea din urmă ar fi detaliat luate în consideraţie *CLC*-ul ar deveni un *circuit*. Deci abordarea unui *CLC* ca *circuit* sau *formulă* depinde de "porţile elementare" luate în consideraţie.

### 3.6.1 Circuitul logic Universal

La sfîrşitul acestui capitol întrevedem posibilitatea gîndirii unui *CLC* care să poată calcula *toate* funcţiile logice imaginabile. Teoretic acest circuit ar fi cel reprezentat în figura 3.55 sub forma unui *ROM* cu $2^{n+q}$ locaţii de $m$ biţi. Deci un *ROM* de $m \cdot 2^{n+q}$ biţi ar putea fi un circuit care calculează $2^q$ funcţii distincte, operînd cu cuvinte de intrare de $n$ biţi şi dînd rezultate de $m$ biţi. Poate fi un astfel de circuit realizat? Dacă răspunsul ar fi afirmativ s-ar putea ca acest curs să se încheie în acest punct. Din păcate, datorită *dependenţei exponenţiale* a capacităţii (complexităţii) *ROM*-ului de numărul $n+q$ de intrări, o astfel de soluţie nici pe departe nu este posibilă.

Circuitul logic Universal ar avea

$$D_u(n+q) = O(1),$$
$$S_u(n+q) = O((n+q)2^{n+q}).$$

Nu s-ar putea ca renunţînd la o adîncime constantă (independentă de numărul de intrări, $n+q$) să obţinem scăderea semnificativă a complexităţii? Pentru o serie de *CLC*-uri particulare acest lucru este posibil (a se vedea Capitolul *8*).

Circuitul logic Universal ar avea înmagazinate în structura lui (memorate în *ROM*) valorile de ieşire pentru toate funcţiile şi toate valorile de intrare. Acest circuit ar "şti totul"! Nu am avea nevoie pentru calculul valorii unei funcţii, de nici un algoritm, ideea de mecanism de calcul ar fi lipsită de sens. Totul s-ar reduce la cititul într-o tabelă imensă. Inteligenţa maşinilor digitale, atît cît poate fi numită astfel, nu ar mai avea sens. Numai duritatea dependenţei exponenţiale ne determină să căutăm şi alte procedee. Ele vor constitui substanţa capitolelor următoare.

Evaluarea cantitativă ne determină să acceptăm faptul că Circuitul logic combinaţional Universal *este lipsit de sens* în perspectivă practică. Îl putem gîndi, dar sîntem în acelaşi timp siguri că nu-l vom putea realiza niciodată. Ar fi fost şi păcat ca totul să fie atît de simplu.

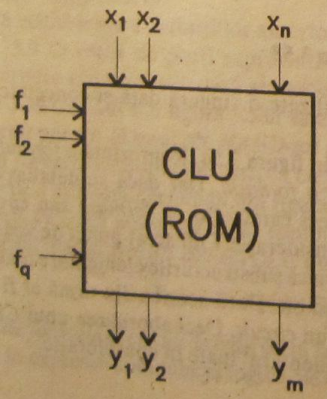



**Figura 3.55**



### 3.6.2 Teorema lui Spira

Anumite moduri de definire a funcţiilor logice, uneori cele recursive, impun structuri de complexitate mică dar de adîncime supărător de mare.

**Exemplul 3.40** Sumatorul din figura 3.22*b* este construit după principiul că un sumator de $n+1$ biţi se construieşte pornind de la un sumator de $n$ biţi ce se cascadează cu unul de *1* bit. Rezultă un *CLC* de tip *formulă* cu $D_\Sigma(n) = O(n)$. □

**Exemplul 3.41** O variantă posibilă pentru a construi un *XOR* cu $n$ intrări este cea de a conecta în cascadă $n-1$ astfel de circuite cu *fan-in* = 2, ca în figura 3.56. Şi în acest caz $D_{XOR}(n) = O(n)$ iar $S_{XOR}(n) = O(n)$. Pentru $n = 8$ există varianta din figura 3.18. Folosind acest principiu de structurare se obţine la aceeaşi complexitate o adîncime $D_{XOR'}(n) = O(\log n)$. □

Există un procedeu general prin care o structură arborescentă (*CLC* de tip formulă) să poată fi transformată de la o adîncime $O(n)$ la una de tip $O(\log n)$, aşa cum ne sugerează exemplul anterior? Teorema ce urmează ne arată că da, oferindu-ne şi procedura de reconfigurare.

**Propoziţia 3.21** Orice *formulă* cu $D(n) = O(n)$ poate fi transformată într-alta cu $D(n) = O(\log n)$. □ (Aceasta este o formă simplificată a teoremei lui Spira.)

*Demonstraţia* ne oferă şi procedeul practic de realizare a transformării. Vom considera cazul limită al $n-1$ structuri $CLC_i$ cu două intrări ce formează un *CLC* tip formulă cu $n$ intrări, $x_1, x_2, \ldots, x_n$ (figura 3.57). Fiecare $CLC_i$ are o singură ieşire. Regula după care se realizează micşorarea adîncimii, este dată în reprezentarea din figura 3.58 unde

- primele $n/2-1$ circuite prin ieşirea lui $CLC_{n/2-1}$ generează semnalul de selecţie la un MUX;

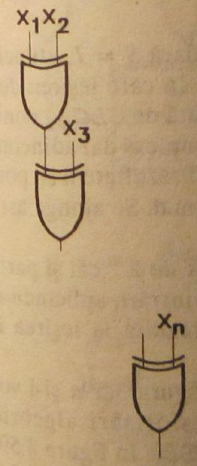



**Figura 3.56**

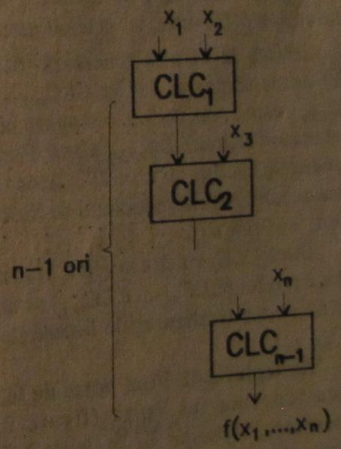



**Figura 3.57**

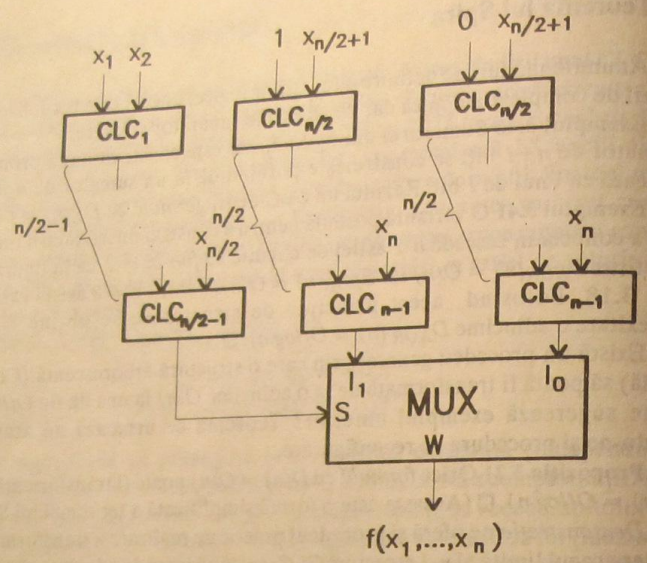



**Figura 3.58**

- ultimele *n/2* circuite sînt dublate iar pe intrarea ce provenea de la $CLC_{n/2-1}$ se aplică *1* la un grup si *0* la celălalt;

- grupul care are pe intrare *1* are conectată ieşirea circuitului $CLC_{n-1}$ la intrarea $I_1$ a *MUX*-ului, iar celălalt este conectat la $I_0$.

Ieşirea *MUX*-ului generează tot $f(x_1, ..., x_n)$ deoarece dacă $S = 1$, atunci la ieşire apare semnalul dat de $CLC_{n/2} ... CLC_{n-1}$ pentru cazul în care ieşirea de la $CLC_{n/2-1}$ este *1*, iar dacă $S = 0$ apare la ieşire valoarea generată de $CLC_{n-1}$ pentru cazul în care ieşirea $CLC_{n/2-1}$ este *0*. Formula a devenit mai complexă dar adîncimea s-a redus de la $O(n)$ la "$O(n/2)$". Celor trei grupuri de *CLC*-uri rezultate li se poate aplica aceeaşi regulă reducîndu-se încă o dată adîncimea, ş.a.m.d. Se ajunge astfel la $D(n) = O(\log n)$

Dacă $CLC_i$ au cîte $m$ ieşiri atunci se utilizează un *MUX* de $2^m$ căi şi partea de jos a reţelei ($CLC_{n/2} ... CLC_{n-1}$) se multiplică de $2^m$ ori pe intrări, aplicîndu-se, pe rînd, toate configuraţiile binare de $m$ biţi ce ar putea rezulta la ieşirea lui $CLC_{n/2-1}$. □

**Exemplul 3.42** Vom porni de la *CLC*-ul *formulă* din figura 3.59*a* şi-i vom aplica teorema lui Spira (figura 3.59*b*). Urmează transformări algebrice elementare: *Propoziţia 3.1* în figura 3.59*c*, relaţia $A' \oplus B = (A \oplus B)'$ în figura 3.59*d*, explicitarea structurii *MUX*-ului în figura 3.59*e* şi evidenţierea *XOR*-ului în ultima etapă. Se transformă astfel un arbore degenerat într-unul echilibrat. □

Cititorul poate aplica aceeaşi procedură unei reţele de *AND*-uri sau *OR*-uri.



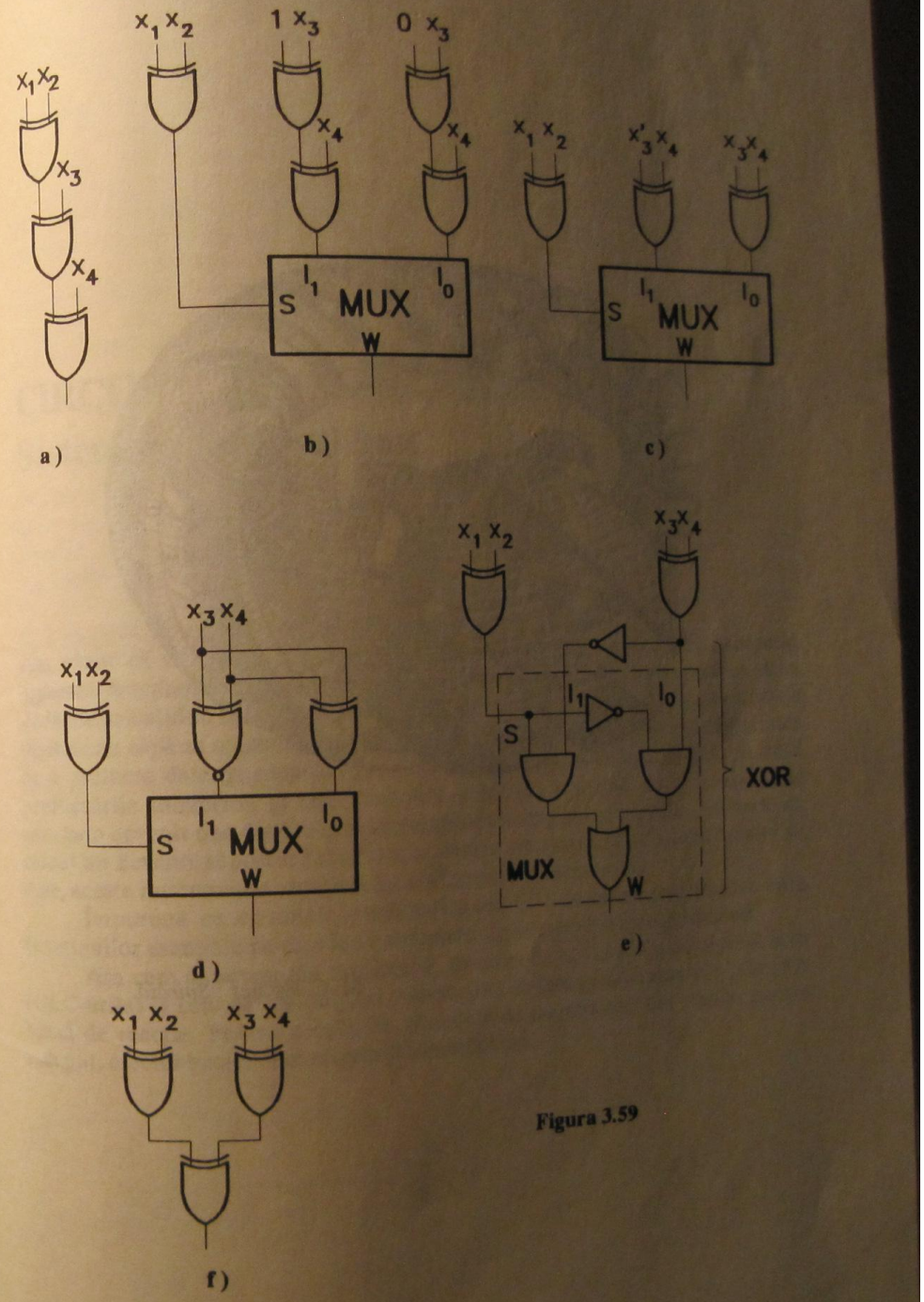
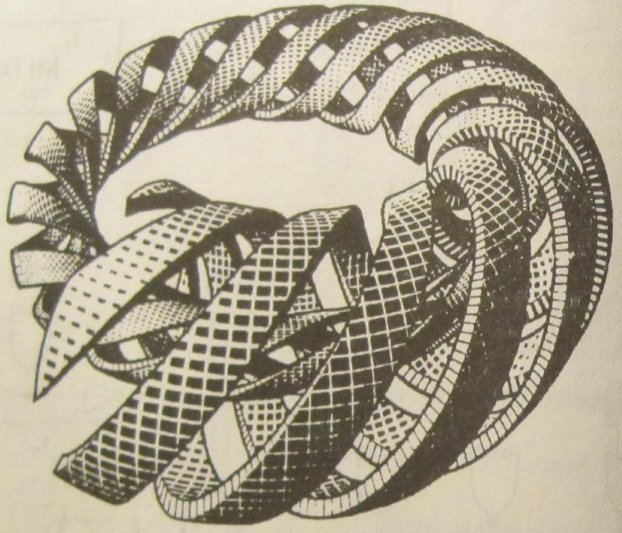



**Figura 3.59**



M. C. Escher : *Spirală.*

4

# CIRCUITE DE MEMORARE
## Sisteme de ordinul unu

Am văzut că circuitele logice combinaţionale pot rezolva, cel puţin teoretic, *aspectele funcţionale* într-un sistem digital. Iniţial sistemele de acest tip erau strict dedicate calculului. În timp s-a produs o reorientare majoră prin faptul că sistemele digitale au căpătat o destinaţie suplimentară: ele au dobîndit, pe lîngă capacitatea de a prelucra date, şi *capacitatea de a stoca date*. După ce s-au impus, pe lîngă prelucrările numerice, şi cele simbolice, *stocarea, manipularea şi transmiterea textelor* a devenit una din funcţiunile esenţiale ale sistemelor digitale. Sistemele de calcul au devenit suportul fizic esenţial pentru realizarea şi utilizarea *bazelor de date*, aceste instrumente esenţiale ale civilizaţiei contemporane.

Împreună cu circuitele combinaţionale, cele de memorare stau la baza funcţiunilor esenţiale pe care le au sistemele digitale: *prelucrarea* şi *stocarea*.

Aşa cum anunţam din capitolul *2*, trecerea de la sistemele de ordinul zero (*CLC*-urile) la cele de ordinul unu (memoriile) se face prin structuri ce posedă o buclă de reacţie. Pentru a delimita circuitele de memorare, vom studia, pentru început, efectul buclei în configuraţii semnificative.

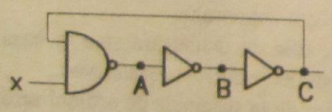
## 4.1 Efectul buclei de reacţie peste un circuit logic combinaţional

### 4.1.1 Structuri oscilante

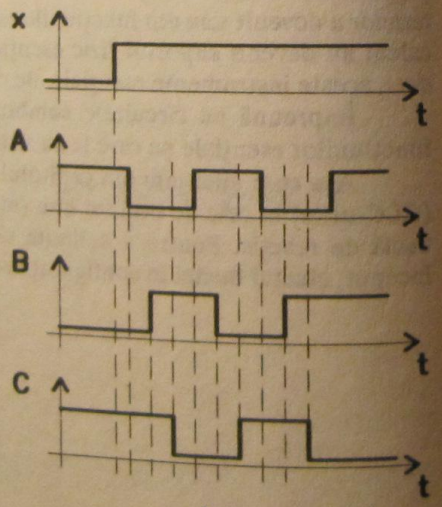
Efectul reacţiei este diferit în funcţie de paritatea sau imparitatea numărului de nivele logice inversoare peste care se închide. Vom studia, pentru început, efectul acesteia într-o buclă cu un număr impar de nivele inversoare.

• **Structura oscilantă** din figura 4.1*a* reprezintă forma simplificată a unei bucle ce se poate închide peste un număr de trei nivele inversoare. Bucla se amorsează la comutarea semnalului $x$ din *0* în *1*. În sistemele digitale o buclă închisă se deosebeşte esenţial de una dintr-un sistem analogic. De regulă, o variaţie a semnalului, într-un punct al unei bucle analogice, se transmite în orice alt punct cu un defazaj mai mare sau mai mic. Spunem că bucla este continuu închisă. La un sistem logic există posibilitatea ca o tranziţie de foarte mică durată să nu se transmită pe buclă. Fenomenele de stocare a sarcinii în joncţiunea bază-emitor spre exemplu, fac ca, pe durata evacuării ei, efectul unui semnal aplicat bazei tranzistorului să nu fie resimţit în colector. Bucla în acest caz nu este continuu închisă. Pe buclă se propagă semnalele cu întîrzierile date de fiecare circuit logic, numai dacă acestea sînt suficient de lungi.

Formele de undă ce descriu comportamental buclei sînt redate în figura 4.1*b*. Se observă faptul că circuitul generează un semnal dreptunghiular cu o perioadă şi un factor de umplere dictate de întîrzierile circuitelor. Tranziţia în *1* a lui $x$ se propagă la cealaltă intrare a *NAND*-ului sub forma unei comutări în sens invers, din unu în zero. Aceasta din urmă se reîntoarce în acelaşi punct inversată după ce mai parcurge o dată bucla, ş.a.m.d.. Dacă $x = 0$ bucla este dezamorsată, poarta fiind închisă.




a)




b)

Figura 4.1



• **Amorsarea oscilaţiei** este realizată tot de semnalul $x$. Valoarea $x = 1$ permite închiderea buclei, dar tranziţia din zero în unu a permis amorsarea procesului de generare a cărui "fază" este astfel controlată.

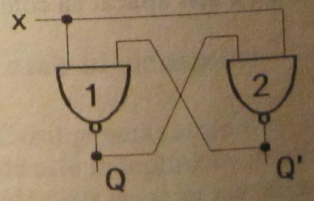
• **Stabilitatea parametrilor** semnalului dreptunghiular generat este dependentă de timpii de propagare prin porţile ce constituie bucla. Din punct de vedere tehnologic aceşti timpi, în majoritatea tehnologiilor, nu pot fi riguros controlaţi. (Un control mai bun se realizează în tehnologia *CMOS*.) Dependenţa de exemplar, de temperatură sau de îmbătrînirea circuitelor face ca astfel de circuite să fie folosite ca generatoare de semnal dreptunghiular numai în cazuri de excepţie.

### 4.1.2 Structuri care se blochează

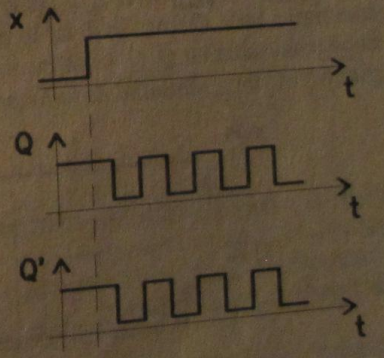
Buclele ce posedă un număr par de circuite inversoare vor avea un alt comportament, chiar dacă, într-o primă instanţă, pe un model teoretic, ideal, se vor comporta la fel.

• **Structurile perfect simetrice** sînt structuri ideale cu un comportament ce nu poate fi întîlnit la nici un circuit real. Să ne imaginăm următorul experiment: circuitului din figura 4.2*a*, realizat cu două *NAND*-uri *identice* (evident, numai teoretic) i se aplică semnalul $x$. Dacă $x = 0$ atunci cele două ieşiri au aceeaşi valoare, *1*. Cînd $x$ comută în *1*, bucla se închide şi, datorită întîrzierilor identice, cele două ieşiri încep să comute identic (figura 4.2*b*).

Explicaţia este relativ simplă. Circuitele fiind identice, fenomenul se produce ca şi cum fiecare *NAND* are conectată la intrare propria ieşire. Deci bucla se închide, ideal, numai peste un singur circuit. Avem două bucle identice formate dintr-un număr impar de inversoare, deci comportamentul va fi cel descris în paragraful anterior.




a)




b)

Figura 4.2

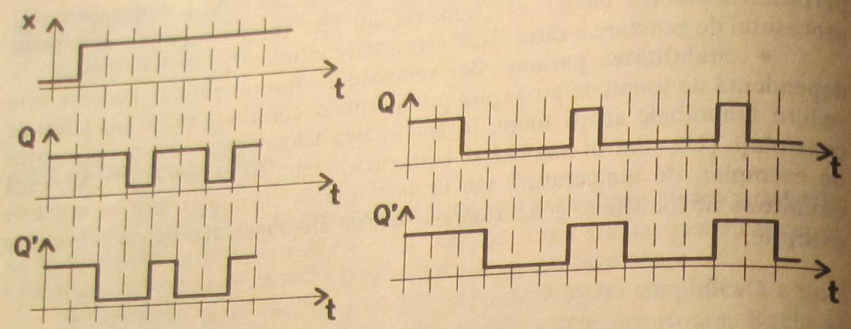



Figura 4.3

• **Structura parţial simetrică** va avea un comportament asemănător. În continuare $t_{pLH} = t_{pHL}$ la ambele *NAND*-uri, dar $t_{p1} > t_{p2}$. Vom considera $t_{p1} = 2t_{p2}$. Rezultă formele de undă din figura 4.3*a*, care certifică prezenţa oscilaţiilor după tranziţia lui *x* în *1*.

Formele de undă din figura 4.3*b* sînt desenate în cazul în care toţi cei patru timpi de propagare sînt distincţi: $t_{pHL1} = t$, $t_{pHL2} = 2t$, $t_{pLH1} = 3t$ şi $t_{pLH2} = 4t$. Simetria ce se mai păstrează este dată de faptul că $(t_{pLH} - t_{pHL})$ este acelaşi pentru ambele porţi.

Va trebui să luăm în consideraţie o configuraţie de timpi de propagare în care să nu mai putem găsi nici o simetrie.

• **Structura complet asimetrică** va fi cea care va avea un comportament net distinct de cele anterioare. Vom lua în consideraţie două *NAND*-uri caracterizate prin: $t_{pHL1} = t$, $t_{pLH1} = 4t$, $t_{pHL2} = 2t$ şi $t_{pLH2} = 3t$. La intervalul $t$ (figura 4.4) de la comutarea lui *x*, *Q* trece în *0*, iar la interval de $2t$ şi *Q'* trece tot în *zero*. Apoi la $3t$ după comutarea lui *Q*, *Q'* trece iar în *1*, comutare la care $NAND_1$ răspunde foarte promt, într-un *t*, *rămînînd definitiv pe 0* pentru că nu a mai apucat să comute în *1*

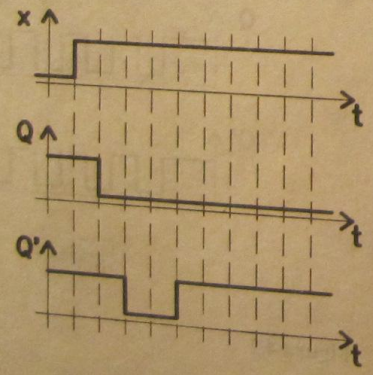



Figura 4.4



drept urmare a comutării în *0* a lui *Q'*, deoarece $t_{pLH1} = 4t$. *NAND*-ul al doilea răspunde mai întîi tranziţiei pozitive a lui *Q'*, nemaiputînd reacţiona la cea negativă. Astfel procesul oscilant este imposibil, circuitul blocîndu-se în starea $Q = 0$ şi $Q' = 1$.

Numai în momentul în care *orice simetrie a fost înlăturată* circuitul s-a blocat într-o anumită stare. Starea normală a unui astfel de circuit este cea în care $Q = 0$ sau $Q = 1$, în funcţie de timpii de propagare asociaţi porţilor din care este constituit. *Experienţa arată că o altă posibilitate nu există* cu toate că teoretic pot fi avute în vedere şi procese instabile.

• **"Experimentul" anterior probează imposibilitatea identităţii.** Într-adevăr, dacă anumite identităţi între timpii de propagare sau relaţii între timpii de propagare ar fi în realitate îndeplinite, atunci în numărul incomensurabil de mare de astfel de circuite existente, o parte, oricît de mică, ar manifesta tendinţa de instabilitate. În realitate astfel de structuri sînt realizate în condiţii foarte bune de reproductibilitate a parametrilor. Nici un inginer nu se preocupă de asimetrizarea structurii şi totuşi ea iese întotdeauna asimetrică. Prin aceasta se poate face o probă destul de puternică pentru imposibilitatea identităţii ca realitate. (O probă absolută este imposibilă.)

Reuşim în acest moment să aducem un puternic argument privind imposibilitatea reală a principiului identităţii (*Propoziţia 3.1*) ce stă la baza logicii formale. Dacă acest principiu ar fi confirmabil într-o realitate oarecare, circuitele de memorie ce fac obiectul acestui capitol, nu ar fi posibile în forma simplă în care vom vedea că există.

Sistemele de ordinul unu vor fi fundamentate pe structuri ce neagă principiul ce stă la baza descrierii formale a sistemelor de ordinul zero. Această contradicţie formală are loc în domeniul pur teoretic, în realitatea sistemelor inginereşti ea nu acţionează distructiv ci numai limitativ.

## 4.2 Latch-uri

### 4.2.1 Latch-ul elementar

Cazul general al unui circuit digital cu memorie este reprezentat în figura 4.5, unde este îndeplinită condiţia ca *pentru orice configuraţie binară $X_i$ bucla să nu se închidă decît peste un număr par de nivele logice inversoare*. Din punct de vedere practic o astfel de abordare este incomodă pentru că se distinge greu funcţia de prelucrare de cea de memorare.

• **Segregarea funcţională** este o tendinţă firească prin care în sistemele digitale funcţia de memorare se delimitează net de cea de prelucrare. Structurile fizice elementare de ordinul unu (cu o singură buclă de reacţie) sînt astfel gîndite încît să îndeplinească *numai funcţia de memorare*. Astfel, într-un sistem digital

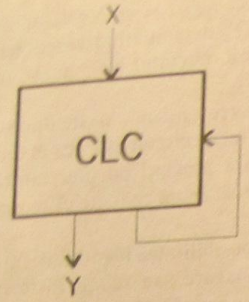



Figura 4.5

concret, zonele de memorare vor conţine numai celule de memorare, net distinct de zonele de prelucrare combinaţională. Acest mod de gîndire asigură o proiectare mai simplă, o punere la punct şi o întreţinere mai eficiente. Din punctul de vedere al evoluţiei tehnologice asigură o dinamică mai mare a performanţelor deoarece permite o concentrare mai mare asupra unor probleme specifice.

Astfel, în continuare vom studia numai celule elementare de memorare sau sisteme de memorie strict orientate numai către această funcţie.

• **Structura latch-ului elementar** este reprezentată în figura 4.6 prin cîteva variante. Cea mai uzuală este cea din figura 4.6*a*. Reacţia închizîndu-se peste un număr par de nivele inversoare, circuitul are *două stări stabile*, fapt pentru care face parte din categoria *circuitelor basculante bistabile*. Într-o stare stabilă, $Q = 0$, iar în cealaltă, $Q = 1$. Cealaltă ieşire, după cum indică şi notaţia, are de fiecare dată valoarea negată.

Comutarea stării se face prin semnalele $S'$ (*set*) şi $R'$ (*reset*) ce sînt *active pe zero*, în sensul că determină comutarea prin valoarea zero. Dacă $S' = 0$, atunci $Q = 1$ iar dacă $R' = 0$, atunci $Q' = 1$. Dacă $S' = R' = 1$, atunci circuitul îşi păstrează starea curentă dobîndită la ultima tranziţie temporară a lui $S'$ sau $R'$ în *zero*.

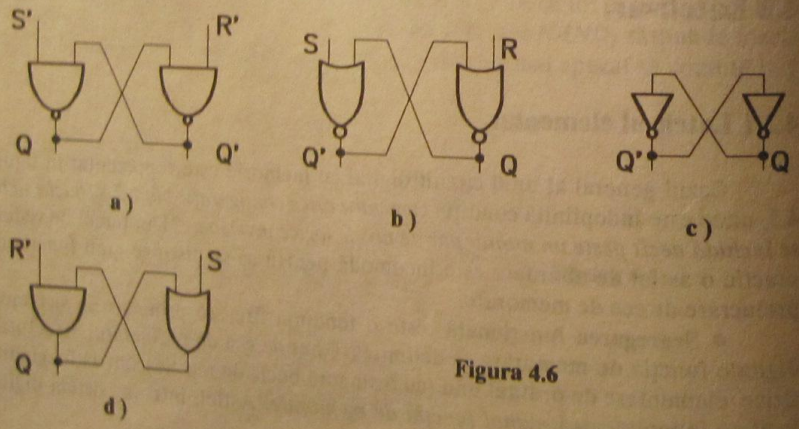



Figura 4.6



Activarea ambelor intrări este o acţiune lipsită de sens, deoarece drept urmare a unei astfel de comenzi latch-ul trece într-o stare ce nu poate fi riguros precizată anterior. În egală măsură ne putem aştepta la comutarea în *1* sau în *0*. Un exemplar dat, într-un context de asemenea dat, va comuta, cu mare probabilitate, întotdeauna în aceeaşi stare, *0* sau *1*, datorită asimetriilor pe care le are sau le resimte.

În figura 4.7 este dat tabelul care descrie funcţionarea latch-ului format din două *NAND*-uri. Pentru $R' = S' = 1$ latch-ul nu comută. Dacă se aplică impulsuri în zero pe $R'$ sau $S'$, atunci circuitul comută în *0* sau în *1*. Dacă se aplică sincron impulsuri pe $S'$ şi $R'$, atunci circuitul are o *tranziţie nedeterminată*.

Latch-ul se comportă ca un circuit de memorare deoarece efectul tranziţiei temporare în zero a intrării este menţinut prin starea bistabilului. Prin $R'$ "scriem" *0* în bistabil, iar prin $S'$ scriem *1*. Zeroul de pe intrarea $S'$ se propagă pe buclă şi se reîntoarce la cealaltă intrare a aceluiaşi *NAND fixînd efectul*, astfel încît zeroul de pe $S'$ poate să dispară.

*Durata minimă a impulsului de comandă* trebuie să fie suficient de mare pentru ca bucla să se poată închide fixînd (zăvorînd) evenimentul. (Termenul de *latch* vine de la verbul englezesc *to latch* (a încuia, a zăvori).) Pentru circuitele uzuale durata acestui impuls este de ordinul $5 \div 20$ *ns*.

O variantă de latch este reprezentată în figura 4.6*b*, fiind realizată cu circuite *NOR*. Se distinge de circuitul anterior prin faptul că este comandată cu semnale active pe unu. Latch-ul din figura 4.6*c* poate fi comandat pe ieşiri. O tranziţie temporară a potenţialului din $Q$ sau $Q'$ în zero, forţată de un circuit exterior, determină comutarea în *0* sau *1*, nepericlitînd integritatea circuitului deoarece acesta, foarte rapid, comută sub efectul buclei. Ultima variantă de latch, cea din figura 4.6*d*, este mai exotică, avînd un număr de zero nivele logice inversoare pe buclă şi fiind comandat de semnale diferite pentru comutare, unul activ în *zero*, iar celălalt în *unu*.

Ar fi interesant de explicat de ce nu se poate realiza un latch prin cuplarea, într-o configuraţie asemănătoare, a două circuite *XOR*.

• **Deficienţele latch-ului elementar** vor impune următoarele tipuri de structuri, mai sofisticate.

O primă deficienţă a şi fost amintită: comutarea nedeterminată prin activarea ambelor intrări. Acţiunea pe intrare este deterministă iar efectul pe ieşire este imprevizibil. O astfel de corelaţie este inadmisibilă într-un sistem digital. Va fi înlăturată în structurile mai complexe.

| S' | R' | Q | Q' |
|---|---|---|---|
| 1 | 1 | Q | Q' |
| 1 | ⊓ | 0 | 1 |
| ⊓ | 1 | 1 | 0 |
| ⊓ | ⊓ | 0/1 | 1/0 |

Figura 4.7

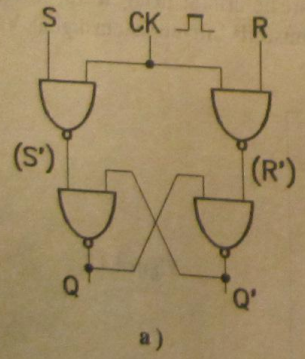
O altă dificultate este dată de faptul că semnalul de pe intrările *S'* şi *R'* determină atît *felul* în care comută latch-ul cît şi *momentul* în care acest fapt se produce. Lipsa distincţiei între *cum?* şi *cînd?* este foarte incomodă în proiectarea sistemelor digitale. Ar fi foarte utilă existenţa separată a unor intrări care să indice modul în care va comuta bistabilul şi a unei intrări pe care să putem impune momentul în care comutarea să se producă.

### 4.2.2 Latch-ul cu ceas

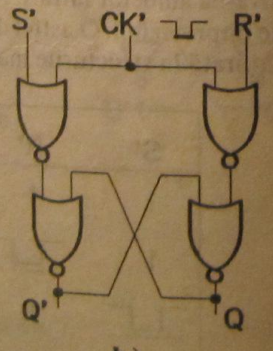
A doua deficienţă anterior semnalată va fi *parţial* înlăturată prin introducerea *latch-ului cu ceas*. O variantă a acestui circuit este reprezentă în figura 4.8*a*. Latch-ului elementar i-au fost adăugate două *NAND*-uri prin care se aplică semnalele de set *(S)* şi reset *(R) condiţionate* de *semnalul de ceas CK* (clock). Numai dacă *CK = 1*, semnalul *S* sau *R* se transmite sub forma *S'* sau *R'* la intrările latch-ului elementar.

Într-o funcţionare coerentă, în intervalul de timp cît *CK = 0* semnalele *S* şi *R* iau valorile dorite, apoi rămîn fixe şi comută ceasul în *1* acţionînd bascularea latch-ului. Spunem că *palierul activ* al ceasului este cel de *unu*. (Pentru latch-ul cu ceas din figura 4.8*b* palierul activ al ceasului este cel de *zero*.)

Figura 4.9 ilustrează corelaţia între semnalele *S, R, CK* şi *Q*. Anterior primului *palier activ* al ceasului, *S* comută în *1* şi odată cu venirea acestuia este determinată tranziţia în *1* a lui *Q*, care anterior era în *0*. Înaintea următorului palier activ, *S* redevine *0* şi *R* comută în *1*, astfel că cel de-al doilea impuls de ceas determină comutarea latch-ului în *0*. Pe durata celui de-al treilea palier activ, *S* comută temporar în *1* determinînd tranziţia bistabilului în *1 (Q = 1)*. În acest al treilea caz comutarea nu a mai fost determinată de tranziţia ceasului. Rezultă că decuplarea dintre *cum?* şi *cînd?* nu este perfectă în cazul acestui bistabil. Pe durata palierului activ latch-ul este *transparent* la modificările de pe *S* sau *R*, el poate comuta drept urmare a tranziţiilor uneia din intrări ce indică *cum?* se modifică



a)



b)

**Figura 4.8**



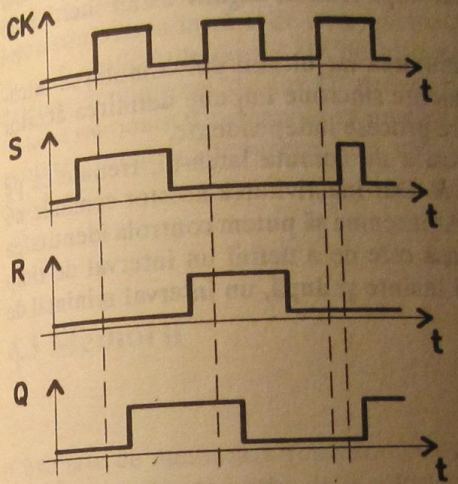



**Figura 4.9**

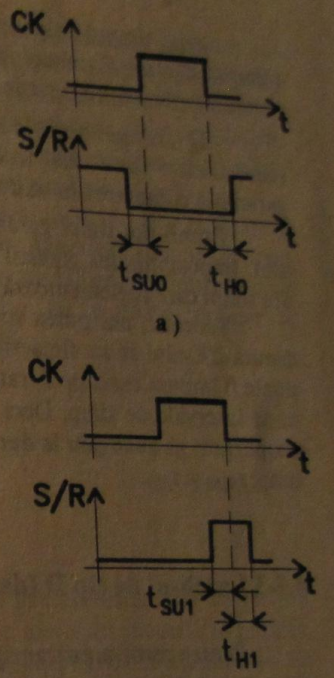



**Figura 4.10**

starea. Decuplarea între *cînd?* şi *cum?* are loc numai dacă *S* şi *R* comută, cînd este cazul, numai în intervalul de timp dat de palierul inactiv al ceasului.

De asemenea, la acest tip de bistabil nu este rezolvată problema tranziţiei nedeterminate, drept urmare a activării sincrone a celor două intrări.

● **Timpii critici de corelaţie între intrările de comandă şi ceas** sînt $t_{SU}$ (timpul de *set-up*) şi $t_H$ (timpul de *hold*). În jurul tranziţiilor ceasului există un interval de timp critic în care intrările *S* şi *R* de anumite valori trebuie să fie stabile.

**Definiţia 4.1** *Timpul de set-up* (timpul de prestabilire) este intervalul de timp dinaintea unui front în care celelalte semnale trebuie să fie stabile. □

**Definiţia 4.2** *Timpul de hold* (timpul de menţinere) este intervalul de timp ce urmează tranziţiei unui front, în care celelalte semnale trebuie să fie stabile. □

Pentru latch-ul din figura 4.8*a* se pot defini următorii timpi, conform reprezentărilor din figura 4.10:

- $t_{SU0}$, timpul de set-up al lui *S* sau *R* în *zero* faţă de frontul pozitiv al ceasului (intervalul de timp înaintea frontului activ în care *S* sau *R* trebuie să fie *zero* pentru ca să nu fie acţionată comutarea bistabilului);

- $t_{H0}$, timpul de hold al lui *S* sau *R* în zero faţă de frontul negativ al ceasului (cît timp mai trebuie menţinut *S* sau *R* pe *zero* după dispariţia impulsului de ceas pentru a fi siguri că latch-ul nu a fost acţionat de acest impuls);

- $t_{SUI}$, timpul de set-up în *unu* al lui *S* sau *R* față de frontul negativ al ceasului (timpul cît *S* sau *R* trebuie să aibă valoarea unu înainte de frontul negativ pentru ca bistabilul să comute sigur în *unu* sau în *zero*);

- $t_{HI}$, timpul de hold în *unu* al lui *S* sau *R* față de frontul *negativ* al ceasului (timpul cît *S* sau *R* mai trebuie menținut după frontul *negativ* astfel încît să fie garantată comutarea în unu sau în zero).

Primii doi timpi garantează necomutarea iar ultimii doi comutarea latch-ului. Imposibilitatea gîndirii unor evenimente sincrone impune definirea acestor doi timpi care reglementează relația dintre procese independente.

Teoretic, am putea spune că pentru a nu comuta latch-ul, trebuie ca pe durata *CK*-ului să nu fie activ nici *S* nici *R*. Dar inactivitatea acestor semnale nu poate fi impusă exact pe durata ceasului. Ar însemna să putem controla identitatea unor intervale de timp. Deci, singura șansă este de a defini un interval de timp acoperitor, să adăugăm la durata *CK*-ului înainte și după, un interval minimal de timp, $t_{SU0}$ și $t_{H0}$.

### 4.2.3 Latch-ul de tip D (date)

Pentru rezolvarea tranziției nedeterminate se poate impune o soluție brutală și anume *restricționarea evoluției intrărilor S și R* astfel încît $R = S'$, notînd unica intrare, astfel obținută, cu *D*. În figura 4.11*a* este construit un *latch de tip D* pornind de la un latch *RS* cu ceas și generînd semnalul *R* printr-un inversor pornind de la *S*. Simbolul acestui nou bistabil este dat în figura 4.11*b*.

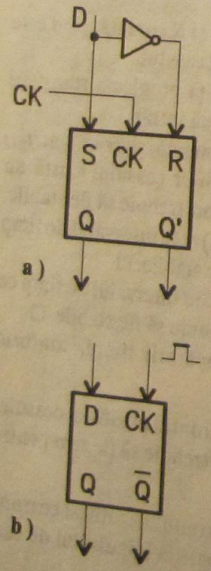



Figura 4.11

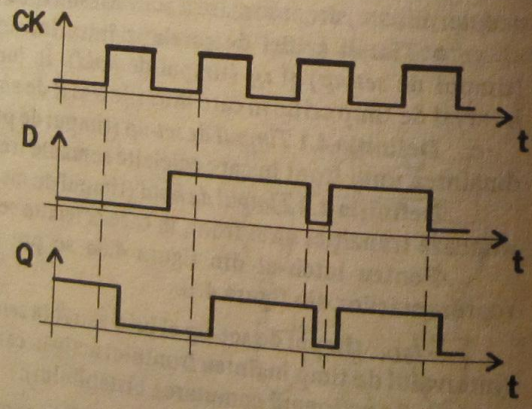



Figura 4.12



Funcționarea latch-ului de tip *D* este sugerată prin formele de undă din figura 4.12. Dacă pe durata palierului activ, *D* nu comută, atunci tranziția latch-ului este controlată de ceas (primele două cicluri de ceas). Comutarea intrării de date (*D*) în intervalul în care ceasul este activ determină, de asemenea tranziția ieșirii. Spunem, ca și în cazul latch-ului *RS*, că latch-ul de tip *D* este *transparent* (simte la ieșire comutarea intrării de date) pe toată durata palierului activ al ceasului.

- **Aplicațiile** acstui tip de bistabil sînt legate de memorarea temporară a datelor, denumirea "de tip *D*" provenind de la *date*. Remarcăm faptul că acest bistabil nu poate fi comandat să nu comute, el urmărește la fiecare impuls de ceas evoluția intrării, deoarece, din cauza inversorului, nu se poate impune ca $S = R = 0$.

## 4.3 Memorii

Circuitele de memorare reprezintă extensia paralel în clasa sistemelor de ordinul unu. În esență este vorba de o colecție de latch-uri, teoretic conectate în paralel, accesul fiind permis nerestricționat, indiferent de poziția latch-lui sau de succesiunea acceselor pentru scriere sau citire. Pe lîngă latch-urile propriu-zise o memorie mai conține și circuitele combinaționale care permit scrierea și citirea în și din acestea.

### 4.3.1 Latch-ul adresabil

Schema de principiu a unui *latch adresabil* este dată în figura 4.13 unde:

- $O_7 ... O_0$ reprezintă ieșirile a opt celule de memorare;

- *D* este intrarea de date;

- $A_2 A_1 A_0$ reprezintă codul (adresa) prin care se selectează scrierea într-una din cele opt celule;

- *CK'* este ceasul care validează (efectuează) scrierea bitului de pe intrarea *D* în celula de memorie selectată prin adresa $A_2 A_1 A_0$.

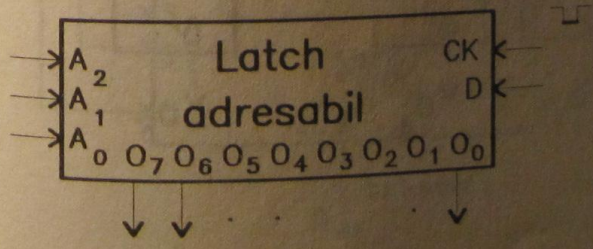



Figura 4.13

Structura internă a latch-ului adresabil este reprezentată în figura 4.14. *DMUX*-ul este folosit pentru distribuirea ceasului către latch-ul selectat în care se va înscrie valoarea de pe *D*, intrarea de date aplicată tuturor latch-urilor. Inversorul de pe intrarea *D* asigură o încărcare minimă circuitului care generează acest semnal. Din acest motiv folosim de la fiecare latch ieşirea $Q_i'$.

• **Atenţie!** Ceasul nu poate fi activat *(CK' = 0)* decît dacă intrările de adrese sînt stabile. Dacă biţii de adresă comută în timp ce *CK' = 0*, atunci pot apare la ieşirile *DMUX*-ului fenomene de hazard datorate comutării a mai mult decît a unui bit de adresă. Tranziţiile parazite astfel produse pot comuta nedorit latch-uri ce în mod normal nu ar trebui să comute. Deci, este obligatoriu ca în intervalul de timp în care *CK' = 1* să fie comutate adresele pe valoarea dorită, urmînd ca apoi ceasul să comute pe palierul activ. În acest fel ieşirile *DMUX*-ului sînt inactivate atunci cînd pe intrarea lui se poate modifica mai mult de un singur bit, evitîndu-se efectele hazardului datorat asincronismului intrărilor. Intrarea de date, *D*, poate comuta şi pe durata palierului activ dacă se respectă timpii de set-up şi hold indicaţi de fabricantul circuitului.

**Exemplul 4.1** Dacă în timp ce *CK' = 0* adresa $A_2 A_1 A_0$ comută din valoarea *010* în *001*, pe lîngă faptul că se dezactivează ieşirea $O_2'$ a *DMUX*-ului şi se activează ferm ieşirea $O_1'$, în mod tranzitoriu, în funcţie şi de relaţia în timp între comutarea lui $A_1$ şi $A_0$ , se poate activa una din ieşirile $O_0'$ sau $O_3'$. Dacă durata tranziţiei parazite este suficient de mare pentru a determina comutarea unui latch, atunci riscăm ca valoarea de pe intrarea *D* să fie înscrisă şi în latch-urile *0* sau *3*, datorită hazardului dat de asincronismul de pe intrările *DMUX*-ului. □

• **Aplicaţia** principală a acestui circuit este legată de memorarea bit cu bit, prin *D*, a unei configuraţii binare care poate deveni accesibilă în totalitate la bornele unei alte structuri pe ieşirile $O_7 ... O_0$ . Scrierea se face *serial* iar citirea poate fi făcută în *paralel*.

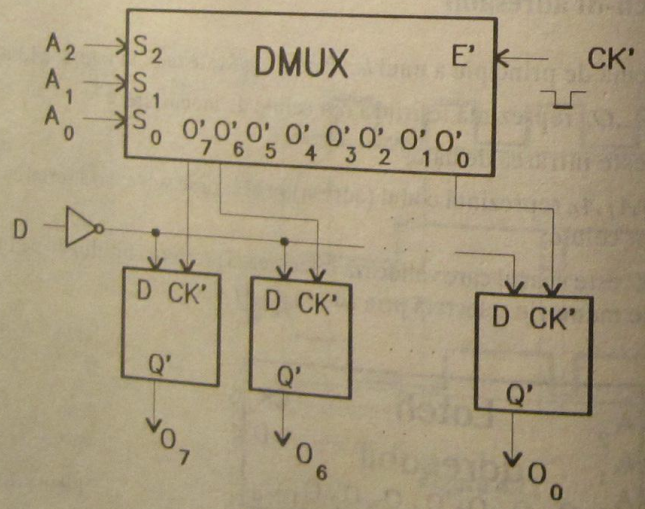



Figura 4.14



• **Limita** esenţială a unei memorii astfel gîndite este aceea dată de citirea în paralel care nu poate fi făcută pentru un număr oricît de mare de biţi, deoarece presupune un număr de conexiuni care, peste anumite valori, este imposibil sau măcar nepractic, de realizat. Pornind de la această limitare vom concepe circuitele din paragraful următor.

## 4.3.2 Structura de principiu a unei memorii cu acces aleator (*RAM*)

O primă variantă pentru un circuit de *memorie cu acces aleator* (random *acces memory - RAM*) este dată în figura 4.15, unde la structura de latch adresabil a fost adăugat un *MUX* ce permite selectarea oricărui bit memorat folosind aceeaşi adresă ca cea care este utilizată pentru scriere. Memoria are $2^{n+1}$ locaţii (latch-uri).

(Termenul de memorie cu acces aleator s-a impus din considerente ce ţin de istoria sistemelor de memorie. Primele structuri fizice folosite la stocarea unui număr mare de biţi s-au bazat pe procese magnetice şi ofereau posibilitatea stocării, pe bandă, tambur sau disc, într-o manieră *serială*, în sensul că pentru a se accesa bitul al *n*-lea trebuia parcursă înregistrarea bit cu bit pînă în dreptul acestuia. Bitul *n* nu putea fi accesat decît după ce a fost accesat bitul *n-1*. În momentul în care s-au putut concepe memorii la care ordinea de accesare a biţilor să fie oarecare s-a

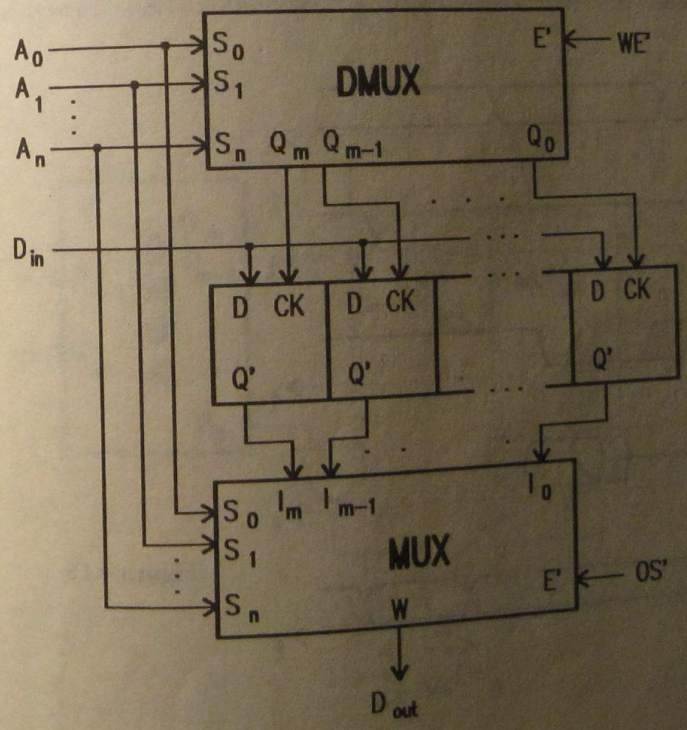



Figura 4.15

realizat un pas foarte mare înainte. Accesul într-o ordine aleatoare oferă facilităţi de utilizare deosebite. Acest fapt a consacrat denumirea de memorie cu acces aleator, *RAM*.)

*DMUX*-ul este folosit numai la scriere iar *MUX*-ul numai la citirea din memoria *RAM* structurată rudimentar în această primă variantă. Corelaţiile esenţiale între semnalele de la bornele acestei memorii sînt prezentate în diagramele din figura 4.16. În momentul $t_1$ începe un *ciclu de citire* prin setarea adresei la valoarea dorită (unii biţi se stabilizează pe zero iar alţii pe unu). Semnalul *OS'* (output select) se poate activa înainte sau după momentul $t_1$, determinînd activarea ieşirii de tip tristate (Anexa 9.32) ce trece în unu sau zero, în funcţie de conţinutul locaţiei selectate. Prin deselecţie (*OS'* = *1*) ieşirea $D_{out}$ trece iarăşi în *starea de mare impedanţă* (*Hi-Z*). Adresa se schimbă în momentul $t_2$, cînd începe un *ciclu de scriere*. După un interval de timp, $t_{AW}$, minim, poate fi activată şi comanda de scriere *WE'* (*w*rite *e*nable) care are o durată, $t_W$, minimă, impusă prin datele de catalog, condiţionată de structura internă a memoriei. Intrarea de date, $D_{in}$, trebuie să fie stabilă un interval de timp, $t_{SU}$, înainte de tranziţia pozitivă a semnalului *WE'*; de asemenea trebuie menţinută un timp, $t_H$, după dispariţia acestui semnal, pentru ca scrierea să fie sigură. Cînd un circuit de memorie este folosit în proiectarea unui sistem trebuie consultată cu atenţie foaia de catalog ce-i descrie funcţionarea şi-i dă "timing"-ul critic. Fiecare tip de memorie şi fiecare firmă posedă caracteristici ce trebuiesc luate în considerare în definirea unei structuri concrete.

**Propoziţia 4.1** *Complexitatea RAM-ului realizat ca o structură liniară de celule* este dată, în principal, de circuitele de acces, *DMUX*-ul şi *MUX*-ul, dacă numărul de locaţii depăşeşte o anumită dimensiune. □

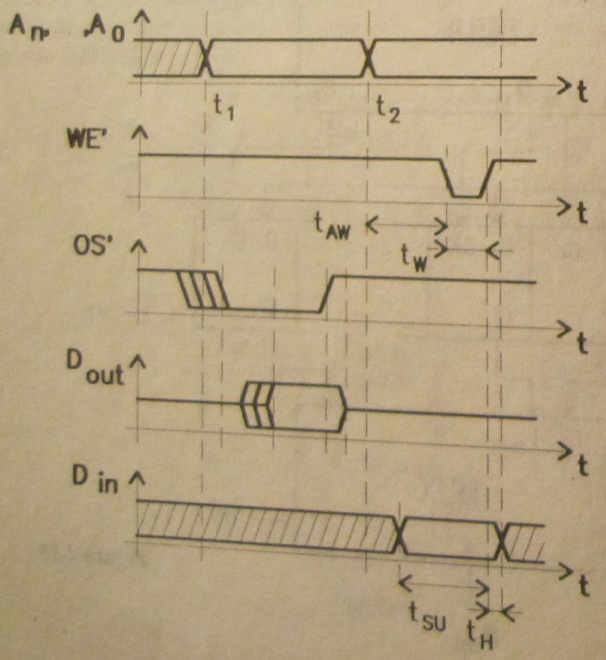



**Figura 4.16**



*Demonstraţia* este imediată dacă se calculează complexitatea *RAM*-ului

$$S_{RAM}(n) = S_{DMUX}(n) + S_{MUX}(n) + S_{LATCH\text{-}uri}(n),$$

unde $n$ este numărul de biţi cu care este dată adresa. Rezultă că

$$S_{RAM}(n) \approx n2^n + (n+2)2^n + k2^n,$$

unde cu $k$ s-a notat complexitatea, constantă, a unui latch. Pentru un $n > n_0$, $S_{RAM}(n)$ va fi dat în principal de primii doi termeni ai sumei. □

Ar fi utilă găsirea unei structuri la care efortul structural major să fie realizat pentru implementarea latch-urilor, ele fiind cele ce dau efectiv funcţia circuitului. Vom căuta o soluţie de acelaşi tip ca în cazul *ROM*- urilor (a se revedea paragraful 3.4.1). Problema suplimentară ce se cere rezolvată este cea a accesului bidirecţional la celula de memorie, în care trebuie să scriem şi pe care trebuie să o citim.

• **MUX-ul bidirecţional** este structura ce va permite realizarea *RAM*-ului ca o matrice bidimensională la care circuitele de acces vor fi nesemnificative în determinarea complexităţii.

Pentru început vom restructura un *MUX* sub forma circuitului din figura 4.17. Un *DMUX*, cu ieşirile active în unu, asigură deschiderea cîte unui *NAND* cu *fan-in* = *2*. Cele $2^n$ *AND*-uri vor fi sumate într-un *OR* a cărui ieşire va fi *W*. *AND*-urile vor putea fi tehnologic realizate sub forma unor porţi de transmisie cu ajutorul unui tranzistor *MOS*. Se obţine astfel, circuitul din figura 4.18, în care, dacă $O_i$ este activată tranzistorul corespunzător va fi deschis şi între $I_i$ şi *W* va fi realizată o *cale bidirecţională*, în sensul că va putea circula curentul în ambele sensuri pentru că tranzistorul se comportă ca o rezistenţă ohmică de mică valoare. Acest circuit se

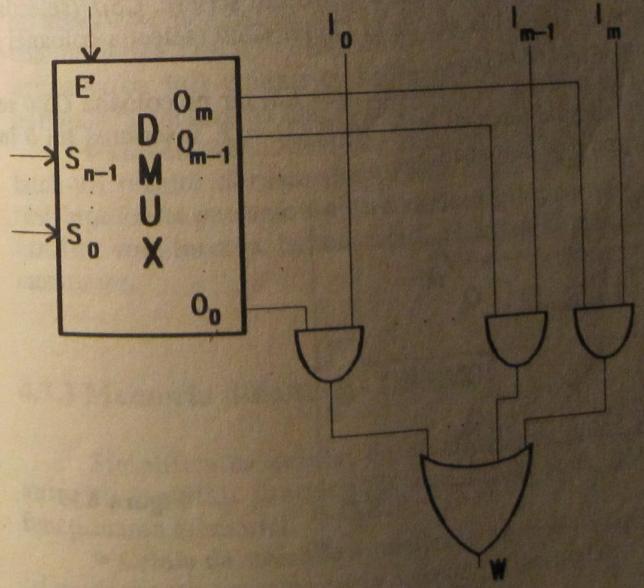



**Figura 4.17**

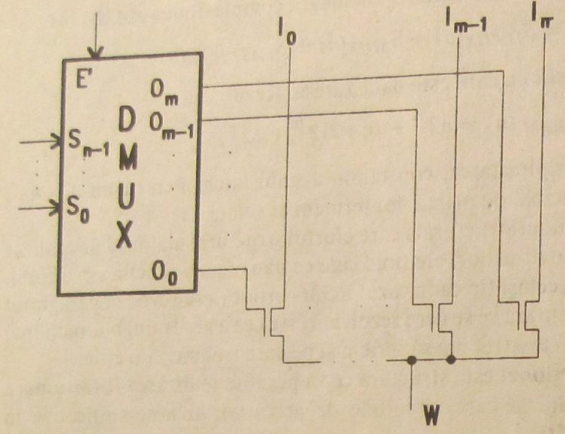



**Figura 4.18**

va putea comporta ca un *MUX* cu intrările $I_i$ şi ieşirea $W$, sau ca un *DMUX* cu intrarea $W$ şi ieşirile $I_i$. Vom denumi acest circuit multiplexor-demultiplexor, *MDX*.

• **Celula elementară** a unei memorii *RAM* porneşte de la latch-ul elementar de tipul celui din figura 4.6c la care se adaugă circuitele ce permit comutarea stării. În figura 4.19 este prezentată o celulă realizată cu 6 tranzistoare *MOS*. Tranzistoarele $Q_1$, $Q_3$ şi $Q_2$, $Q_4$ formează cîte un circuit inversor. Ieşirea $Q_{ij}$ a inversorului format din $Q_1$ şi $Q_3$ este conectată la intrarea celuilalt, a cărui ieşire, $Q_{ij}'$, este legată la intrarea primului. Este format astfel cel mai simplu latch imaginabil. $Q_5$ şi $Q_6$ formează cîte o *poartă bidirecţională* de acces la $Q_{ij}$ şi, respectiv, la $Q_{ij}'$. Porţile de acces sînt deschise de semnalul selecţie de linie $i$, $LS_i$ (*l*ine *s*elect). Liniile verticale permit accesul la celulă pentru citire sau scriere. $CS_j1$ (selecţia coloanei $j$ pentru *1*) permite accesul *la* sau *de la* $Q_{ij}$ iar $CS_j0$ (selecţia coloanei $j$ pentru *0*) permite accesul *la* sau *de la* $Q_{ij}'$.

Pentru a înscrie *1* în celula $ij$ se activează linia $LS_i$ iar pe coloana $CS_j0$ se forţează semnalul *0* care comută latch-ul astfel încît $Q_{ij} = 1$. Înscrierea lui *0* în aceeaşi celulă se face forţînd *0* pe coloana $CS_j1$.

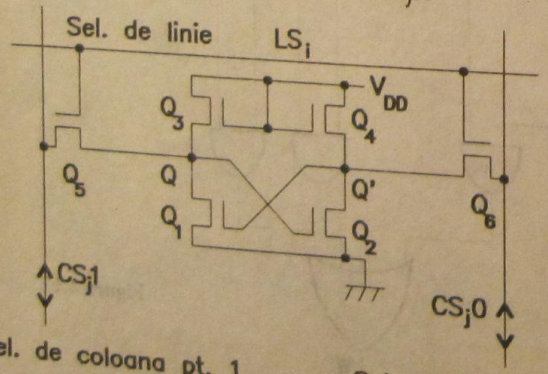



**Figura 4.19**



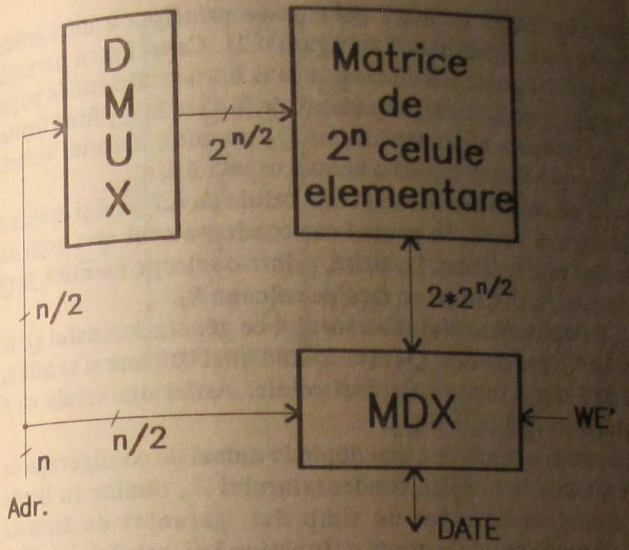



**Figura 4.20**

Citirea conţinutului celulei se face selectînd linia $i$ prin $LS_i = 1$ şi receptînd semnalul de pe coloana $CS_j0$ sau $CS_j1$.

Acest tip de celulă permite realizarea unei *reţele bidimensionale*, liniile $LS_i$ fiind conectate la ieşirea unui *DMUX* iar coloanele $CS_j0$ şi $CS_j1$ la două *MDX*-uri.

• **Realizarea RAM-ului ca o reţea bidimensională** este prezentată în figura 4.20 unde cele $2^n$ (pentru $n$ par) celule de memorie sînt organizate într-o matrice pătrată de $2^{n/2} \times 2^{n/2}$ latch-uri de tipul celui din figura 4.19. Selecţia liniilor este realizată prin *DMUX* iar accesul la bitul căutat pe linie este asigurat de două *MDX*-uri. Într-o astfel de structură, ca şi în cazul *ROM*-ului, ponderea circuitelor de acces este neglijabilă pentru un $n$ suficient de mare. Circuitele de acces au complexitatea $O(n\,2^{n/2})$ faţă de $O(2^n)$ a celulelor de memorare. De asemenea, organizarea într-o matrice bidimensională permite o realizare tehnologică optimală. Structurarea liniară a celulelor ar fi pus probleme dificile de acces.

• Dimensiunea *RAM*-ului fiind dată exclusiv de reţeaua bidimensională de latch-uri rezultă că orice simplificare adusă celulei de memorare elementare se răsfrînge foarte puternic asupra complexităţii circuitului în ansamblu. În paragraful următor vom încerca, în consecinţă, să reducem la minimum structura celulei de memorare.

### 4.3.3 Memoria dinamică - DRAM

Simplificarea celulei de memorare se poate realiza numai sacrificînd anumite facilităţi funcţionale; mai corect spus, adăugînd nişte restricţii în funcţionarea memoriei.

• **Celula de memorie dinamică** se bazează pe faptul că o capacitate parazită îşi poate menţine sarcina puţin alterată un interval de timp suficient de mare dacă

la bornele ei rezistenţele sînt destul de mari. Schema de principiu a unei *celule dinamice cu trei tranzistoare* este reprezentată în figura 4.21. Capacitatea parazită grilă-sursă a tranzistorului $Q_1$ îşi poate păstra tensiunea la borne suficient de puţin modificată dacă tranzistorul $Q_6$ este blocat, impedanţa de intrare în $Q_1$ fiind foarte mare. Rezistenţa canalului deschis al tranzistorului $Q_6$ permite, datorită valorii mici, încărcarea sau descărcarea foarte rapidă a acestei capacităţi, $C_p$.

Scrierea într-o astfel de celulă se face selectînd celula cu $LS_i = 1$ şi aplicînd pe coloana $W_j$' semnalul logic *0* sau *1*. În primul caz condensatorul este golit de sarcină, iar în cel de al doilea este încărcat. La citire, printr-o selecţie identică, în $Q$ vom regăsi semnalul *1*, respectiv, *0*. Citirea se face pe coloana $R_j$.

Condensatorul $C_p$ îndeplineşte rolul inversorului ce genera semnalul $Q'$ în reprezentarea din figura 4.19. Tranzistorul $Q_3$ este comun unei coloane şi se află în josul coloanei $R_j$, dispărînd din structura fiecărei celule. Astfel din celula cu 6 tranzistoare au rămas numai trei: $Q_1$, $Q_5$ şi $Q_6$.

Capacitatea de a memora a celulei nu mai depinde numai de configuraţia de circuit, ci şi de *timp*. Tensiunea la bornele condensatorului $C_p$ rămîne în limite acceptabile alterată, numai un interval de timp dat, garantat de firmele producătoare la *2* ms. Dependenţa de timp a funcţionării celulei justifică denumirea de celulă *dinamică* de memorie spre deosebire de *celula statică de memorie* (figura 4.19) a cărei funcţionare nu depinde de timp. Apar astfel notaţiile *DRAM* pentru memoria *d*inamică şi, uneori, *SRAM* pentru cea *s*tatică.

Pentru a asigura stocarea un interval de timp oricît de mare, într-o memorie dinamică, trebuie ca la cel mult *2* ms conţinutul fiecărei celule să fie *regenerat*. Fiecare locaţie de memorie trebuie citită şi rescrisă pentru a se reface la valoarea nominală sarcina de pe fiecare capacitor $C_p$.

• **Celula de memorie cu un singur tranzistor** este cea care s-a impus ca procedeu standard de realizare a *DRAM*-urilor. Această celulă este dată în figura 4.22. Din structura statică iniţială nu a mai rămas decît $Q_6$ prin care se accesează capacitorul $C_p$. Accesul este controlat prin semnalul $LS_i$ şi realizat efectiv pe coloana $R/W_j$. Schema de principiu a unui cip de memorie *DRAM* este dată în figura 4.23, în care la intersecţia dintre linia $LS_i$ şi coloana $R/W_j$ a fost "pusă lupa" reprezentîndu-se explicit configuraţia fizică. Cu *A* s-a notat circuitul ce asigură

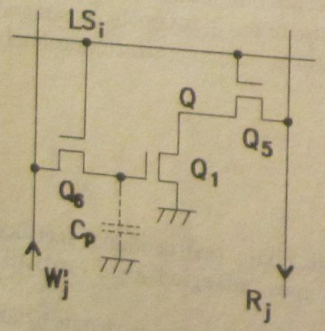



Figura 4.21

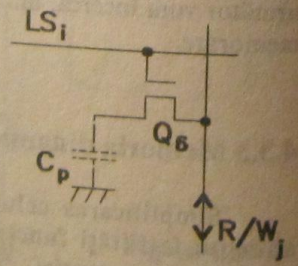



Figura 4.22



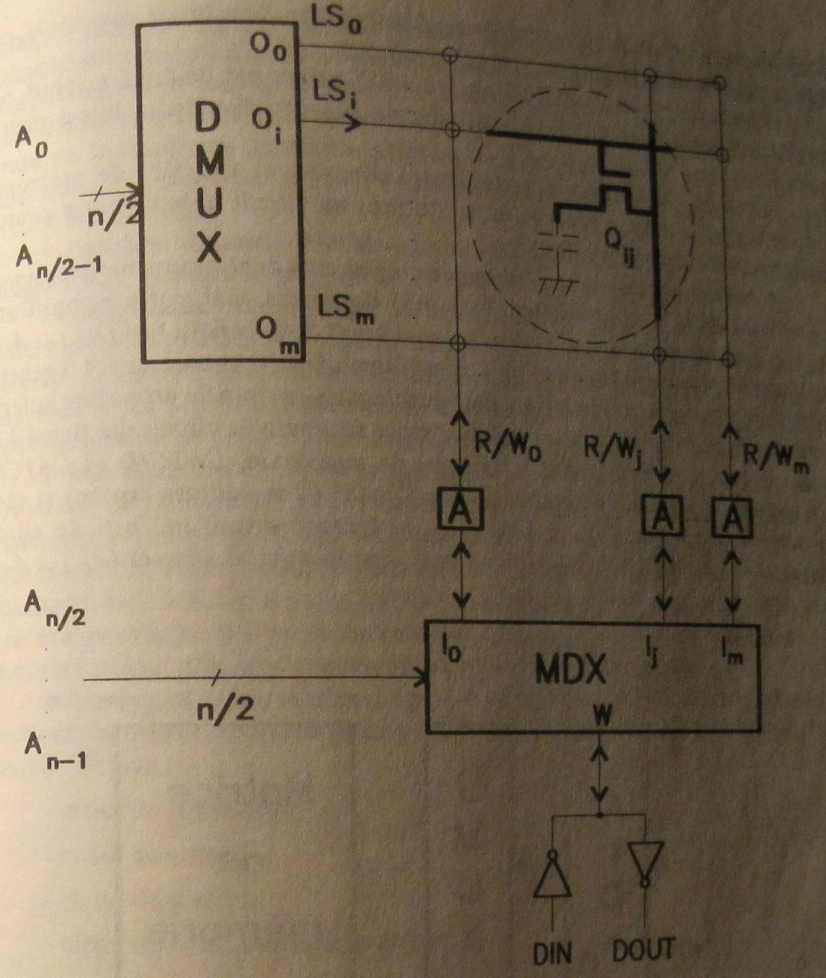



Figura 4.23

generarea nivelului logic *0* sau *1* pornind de la tensiunea de pe condensatorul $C_p$ selectat. Acest circuit realizează, de asemenea, la fiecare selecţie de linie, regenerarea prin reînscrierea în fiecare condensator a valorii citite din acesta. Deci, la citirea celulei de memorie *ij* se realizează regenerarea tuturor celulelor de pe linia *i*. Selecţia bitului din linia *i* se face prin *MDX*. Citirea şi scrierea se fac pe traseele bidirecţionale $R/W_j$, structura amplificatorului *A* permiţînd acest lucru.

• **Structura de MDX a memoriei dinamice** este evidentă la o analiză atentă a schemei interne. Într-adevăr, se poate pune în evidenţă un *MUX* bidirecţional extins pe două niveluri (figura 3.34). Primul nivel este format din *MDX*-ul selectat prin $A_{n/2} \dots A_{n-1}$ care are pe cele $m+1$ intrări ($m=2^{n/2}-1$) tot atîtea *MDX*-uri formate dintr-o coloană de tranzistoare $Q_{0j}, \dots, Q_{mj}$ şi *DMUX*-ul selectat prin $A_0 \dots A_{n/2-1}$. *DMUX*-ul este folosit în comun pentru structurarea celor $m+1$ *MDX*-uri de pe cel de al doilea nivel. Deci avem un *MDX* de $2^n$ căi, structurat pe

două niveluri, prin care se permite accesul la tot atîtea capacitoare parazite $C_p$ care reprezintă elementele propriu-zise de memorare.

Putem spune că *un DRAM este un MDX*, structurat pe două niveluri, prin care se realizează accesul la $2^n$ capacitoare parazite distribuite într-o matrice pătrată.

Dacă se acceptă comportamentul dinamic al funcţiei de memorare (dependenţa de timp a procesului de stocare) un circuit de memorare poate fi realizat ca un sistem de ordinul zero.

• **Adresarea unei memorii dinamice** va ţine cont de structura bidimensională a unui *DRAM*. Se observă faptul că biţii de adresă mai puţin semnificativi ($A_{n-1} \dots A_{n/2}$). ($A_{n/2-1} \dots A_0$) au efect la ieşire pe o cale mai lungă decît ceilalţi biţi ($A_{n-1} \dots A_{n/2}$). Din acest motiv putem spune că biţii cei mai semnificativi ai adresei pot fi generaţi la intrarea structurii cu o întîrziere dată de diferenţa de timp de propagare anterior menţionată. Pornind de la acest fapt s-a impus structura de circuit din figura 4.24, în care apar suplimentar două structuri de memorare, $LATCH_0$ şi $LATCH_1$, comandate de semnalele (active pe zero) *RAS'* (row address strobe) şi *CAS'* (column address strobe). În $LATCH_0$ se încarcă primii $n/2$ biţi de adresă ($A_{n/2-1} \dots A_0$) iar în $LATCH_1$ biţii cei mai semnificativi ai adresei ($A_{n-1} \dots A_{n/2}$).

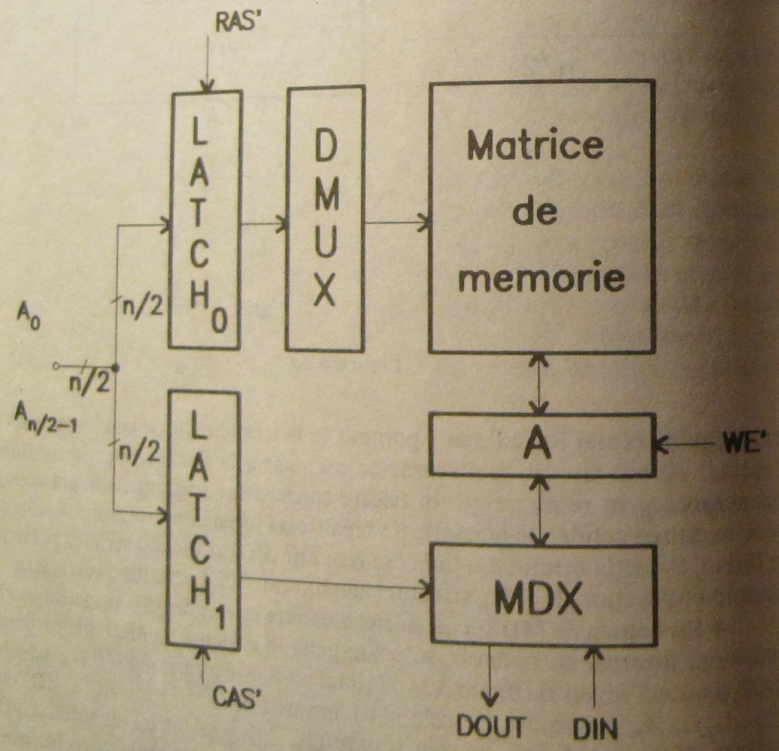



Figura 4.24



Astfel, la intrarea cipului, pentru adresă, sînt suficienţi $n/2$ pini în loc de $n$. Utilizatorul va proiecta astfel sistemul ca la bornele de adresă să fie generaţi întîi primii $n/2$ biti ai adresei ce se încarcă la comanda *RAS'* iar apoi, următorii $n/2$ biţi ce se încarcă în celălalt latch de $n/2$ biţi la comanda *CAS'*. Faptul că în $LATCH_1$ adresa se încarcă mai tîrziu nu afectează eficienţa folosirii circuitului, dacă întîrzierea nu depăşeşte o anumită limită, dată de diferenţa de timp de propagare anterior menţionată.

• **Cipul de memorie DRAM** va arăta, în consecinţă, conform reprezentării din figura 4.25 pentru o capacitate de $2^n$ biţi.

Perfecţionarea tehnologiilor de realizare permite ca la fiecare salt să se obţină o capacitate de memorare de 4 ori mai mare. Acest fapt se datoreşte dublării densităţii pe fiecare dimensiune. Rezultă că fiecare creştere de capacitate implică adăugarea a doi biţi la adresa folosită, deci folosirea a încă unui pin la cipul de *DRAM* pentru că pe fiecare pin se încarcă doi biţi de adresă.

Prima memorie dinamică propusă de producători (*INTEL*, în jurul anului *1970*) a fost de *1K bit*, deci $n = 10$. Numărul de pini pe care se încărca adresa era de *5*. Următoarea a fost de *4K biţi*, deci necesita *6* pini pentru încărcarea adresei. S-a ajuns la *4M bit*, deci $n = 22$, implicînd *11* pini pentru încărcarea adresei. Pînă la capacitatea de *256K biţi* a putut fi folosită aceeasi capsulă de *16* pini, astfel încît, dacă era prevăzut din proiectarea cablajului, se putea mări capacitatea de memorare numai prin implantarea unor circuite mai performante.

• **Timing-ul** la bornele unui cip de memorie dinamică este restricţionat de funcţionarea relativ complexă pe care o are acest tip de circuit. Vom distinge patru tipuri de cicluri:

- ciclul de regenerare,
- ciclul de citire,
- ciclul de scriere,
- ciclul de citire-modificare-scriere.

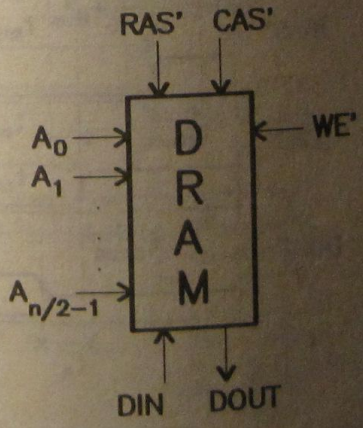



Figura 4.25

*Ciclul de regenerare* asigură reîmprospătarea memoriei, mai bine zis refacerea sarcinii pe condensatoarele $C_p$. Într-un interval de timp de *2 ms* este obligatoriu ca toate celulele să fie reîmprospătate. Procesul de regenerare presupune ca fiecare linie să fie accesată măcar o dată în timp de *2 ms*. Pentru regenerare este deci suficientă o adresă de *n/2* biţi şi semnalul *RAS'*, prin care se selectează o linie al cărei conţinut este citit cu amplificatoarele *A* şi este reînscris în aceleaşi poziţii. Faptul că nu am dat semnalul *CAS'*, cu o adresă oarecare, nu are nici o importanţă pentru că ar fi acţionat în zona *MDX*-ului numai, neafectînd procesul de regenerare. În figura 4.26 este dat ciclul de regenerare, caracterizat, în principal, prin:

- $t_{SU}$ - timpul minim de set-up al adreselor faţă de frontul negativ al *RAS'*-ului pentru a se garanta înscrierea corectă a acesteia în $LATCH_0$;
- $t_H$ - timpul minim de hold al adreselor faţă de acelaşi front;
- $t_{RAS}$ - durata minimă a semnalului *RAS'* pentru a asigura citirea şi reînscrierea corectă a conţinutului liniei selectate.

*Atenţie*! Dacă $t_{SU}$ sau $t_H$ sînt daţi la valoarea minimă egali cu zero, în proiectare va trebui să-i considerăm *strict pozitivi*. Încercînd să-i asigurăm la valoarea zero s-ar putea oricînd, prin abaterile uzuale, ca ei să devină negativi. Zero pentru un inginer nu există! În practica proiectării nu putem folosi decît inegalităţi. Deci, dacă în catalog scrie: $t_{Hmin} = 0\ ns$, atunci ca proiectanţi vom considera că $t_H > 0$ (*strict* mai mare ca zero).

*Ciclul de citire* este reprezentat în figura 4.27. Relaţiile critice sînt cele între comutarea adreselor şi tranziţiile negative ale semnalelor *RAS'* şi *CAS'*. Încărcarea adresei în cip se face în două etape. În prima se aduc biţii de adresă ce selectează linia iar în a doua cei ce selectează coloana din matricea celulelor de memorare. Un set complet al relaţiilor temporale între semnale se extrage întotdeauna din foaia

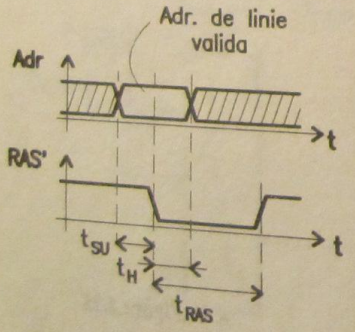



Figura 4.26

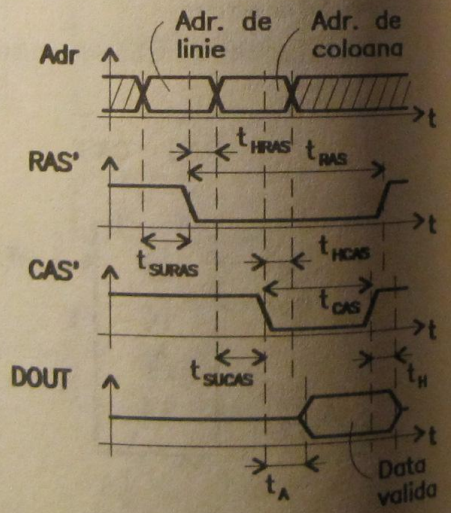



Figura 4.27



de catalog, atunci cînd se proiectează un sistem de memorare. În continuare vom lista numai anumiţi timpi mai semnificativi:

- $t_{RAS}$ - durata minimă a semnalului *RAS'*;
- $t_{CAS}$ - durata minimă a semnalului *CAS'*;
- $t_{SURAS}$ - timpul de set-up al *adresei de linie* faţă de frontul negativ al *RAS'*-ului;
- $t_{HRAS}$ - timpul de hold al adresei de linie faţă de acelaşi front;
- $t_{SUCAS}$ - timpul de set-up al *adresei de coloană* faţă de frontul negativ al *CAS'*-ului;
- $t_{HCAS}$ - timpul de hold al adresei de coloană faţă de acelaşi semnal;
- $t_A$ - *timpul de acces* faţă de *CAS'*, timpul în care ieşirea de tip tristate iese din starea de *Hi-Z* şi generează pe ieşire data adresată;
- $t_H$ - timpul de hold al datelor la ieşire după dispariţia semnalului de *CAS'*.

Ieşirea *DOUT* se activează numai dacă se aplică cipului semnalul *CAS'*. *Aceeaşi atenţionare trebuie făcută pentru stricta respectare a timpilor de set-up şi hold faţă de RAS' şi CAS'*. Pentru preluarea valorii *DOUT* este utilizabil, în aplicaţii curente, frontul pozitiv al semnalului *CAS'*. Nu am figurat semnalul *WE'* deoarece acesta rămîne inactiv, pe *1*.

*Ciclul de scriere* este reprezentat prin ce are specific în figura 4.28 unde:

- $t_{SUD}$ - timpul de set-up al datelor faţă de frontul posterior al semnalului *CAS'*; datele pot fi setate oricînd înainte, dar cel tîrziu cu acest interval de timp înainte de terminarea *CAS'*-ului;
- $t_{HD}$ - timpul de hold al datelor faţă de terminarea semnalului *CAS'*;

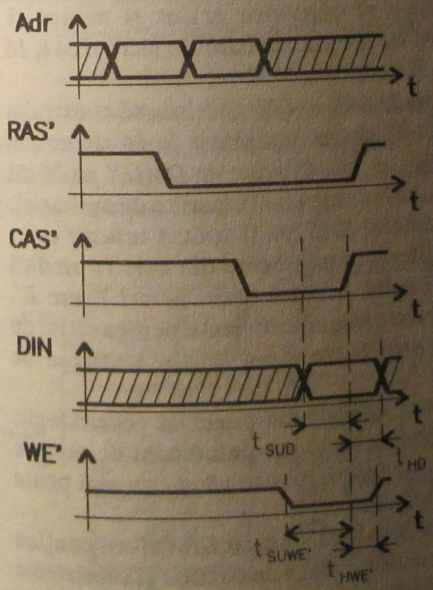



Figura 4.28

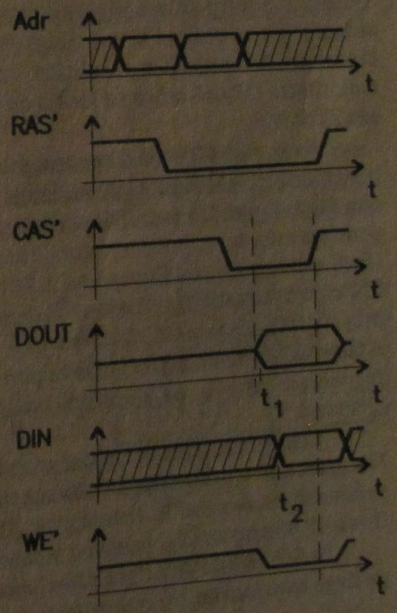



Figura 4.29

- $t_{SUWE'}$ - timpul de set-up al comenzii de scriere faţă de frontul posterior al *CAS'*-ului; comanda *WE'* poate fi activată oricînd mai devreme, chiar înainte de declanşarea ciclului de scriere;

- $t_{HWE'}$ - timpul de hold al comenzii de scriere faţă de frontul posterior al semnalului *CAS'*; poate rămîne activată oricît în continuare.

*Ciclul de citire-modificare-scriere* se dovedeşte foarte util datorită faptului că adresarea memoriei este foarte mare consumatoare de timp. O adresă odată încărcată în $LATCH_0$ şi $LATCH_1$ este bine să fie folosită, atunci cînd se poate, pentru mai mult decît un acces la memorie. Sînt aplicaţii în care se citeşte din memorie, se prelucrează configuraţia binară citită şi apoi *rezultatul prelucrării trebuie reînscris la aceeaşi adresă*. Ciclul în discuţie este util într-un astfel de caz. L-am reprezentat în figura 4.29. În intervalul de timp $t_2$-$t_1$ datele sînt preluate în circuitele de prelucrare, sînt prelucrate şi aduse înapoi la intrările de tip *DIN* pentru a fi reînscrise în *DRAM*. Relaţiile temporale între semnale sînt aceleaşi ca în ciclurile anterioare, exceptînd $t_{RAS}$ şi $t_{CAS}$ care se vor mări corespunzător. Un ciclu *RMW* (*r*ead-*m*odify-*w*rite), cum mai este numit acest mod de funcţionare, durează mai puţin, de regulă, decît un ciclu de citire urmat de unul de scriere.

### 4.3.4 Matrice de cipuri DRAM

Extinderea capacităţii memoriei *DRAM* se face în aceeaşi manieră ca în cazul "memoriilor" de tip *ROM*. (Ne permitem acum să considerăm *ROM*-urile drept "memorii" pentru că am văzut ce este o memorie; *ROM*-ul rămîne un *CLC*, un sistem logic de ordinul zero.) De regulă, sînt utilizate matrici bidimensionale de cipuri *DRAM* pentru a se obţine un spaţiu al adreselor extins şi pentru că majoritatea *DRAM*-urilor au locaţii de un bit iar în aplicaţii curente lucrăm cu *8*, *16* sau *32* de biţi.

**Exemplul 4.2** Dacă dispunem de cipuri de $64K \times 1$ *bit* şi dorim să construim o memorie de $1M \times 32$ *biţi* vom folosi $16 \times 32$ cipuri organizate în *16* rînduri de cîte *32* de cipuri. Va trebui, în schema pur logică, să adăugăm un *DMUX* pe *16* căi pentru a extinde spaţiul adreselor de la *16* (oferit de cip) la *20* (cerute de aplicaţie). (Este utilă revederea *Exemplului 3.34*.) Semnalul *RAS'* va fi aplicat tuturor celor *512* cipuri iar semnalul *CAS'* va fi demultiplexat către fiecare din cele *16* rînduri prin *DMUX*-ul amintit. *CAS'* va juca deci rolul pe care la *ROM*-uri îl are *E'*. Aplicarea în comun a *RAS'*-ului va permite şi o comandă eficientă pentru ciclul de regenerare care se va declanşa odată pentru toate cipurile din matricea de memorie. □

Cele enunţate în exemplul anterior sînt valabile din punct de vedere logic. Din punct de vedere electric, schema va trebui completată ţinînd cont de aspecte suplimentare impuse de dimensiune. Timingul definit pentru *un cip* nu mai poate fi valabil dimensional şi pentru *o matrice de cipuri*.

• **Schema de principiu a unei memorii RAM** realizată cu *DRAM*-uri conţine pe lîngă matricea de circuite de memorare o serie întreagă de circuite şi subsisteme



care permit conectarea cu exteriorul, timingul şi asigurarea fan-out-urilor, acolo unde este cazul.

**Exemplul 4.3** Prezentăm în continuare proiectarea de sistem a unei memorii de $256K \times 16$ *biţi*, folosind circuite *DRAM* de $64K \times 1$ *bit*. Propunem schema bloc din figura 4.30 unde:

- $4 \times 16$ *DRAM* este matricea de cipuri *DRAM* de *64K biţi* organizată în *patru linii* de cîte *16* cipuri;

- *MUX* este format din *8 MUX*-uri de cîte patru căi (cu o cale nefolosită) pentru multiplexarea semiadreselor la intrarea matricei de *DRAM*-uri, după cum urmează:

  • $A_0 A_1 \dots A_7$ pe intrările $I_0$ pentru a fi încărcate cu *RAS'* în $LATCH_0$ din ficcare *DRAM*;

  • $A_8 A_9 \dots A_{15}$ pe intrările $I_1$ pentru a fi încărcate cu $CAS'_i$ în $LATCH_1$ din linia de *DRAM*-uri activată în ciclul curent, $(i=0,\dots,3)$;

  • $R_0 R_1 \dots R_7$ pe intrările $I_2$ pentru a fi încărcate în $LATCH_0$ din fiecare *DRAM* pentru a selecta linia ce se regenerează;

- *REG* este un subsistem ce asigură declanşarea ciclului de regenerare, prin semnalul *R*, şi generarea adreselor la care să se facă regenerarea ($R_7 \dots R_0$); pe calea *AR* primeşte răspunsul de achitare prin care i se comunică faptul că ciclul de regenerare comandat prin *R* a fost efectuat;

- *GC* este generatorul de ciclu care asigură formele de undă necesare efectuării corecte a ciclurilor de citire/scriere sau de regenerare.

Funcţionarea blocului *GC* este descrisă prin formele de undă din figura 4.31 în care sînt prezentate cele două cicluri principale:

- ciclul de citire/scriere (tacturile *0*,...,*7*);

- ciclul de regenerare (tacturile *8*,...,*12*).

Ciclul de citire/scriere se declanşează la comanda *ACC*, prin care se face acces la memorie, iar cel de regenerare prin comanda *R*. În caz de concurenţă, are prioritate comanda *R* activîndu-se comanda *WAIT* prin care cel ce a comandat accesul este pus în aşteptare.

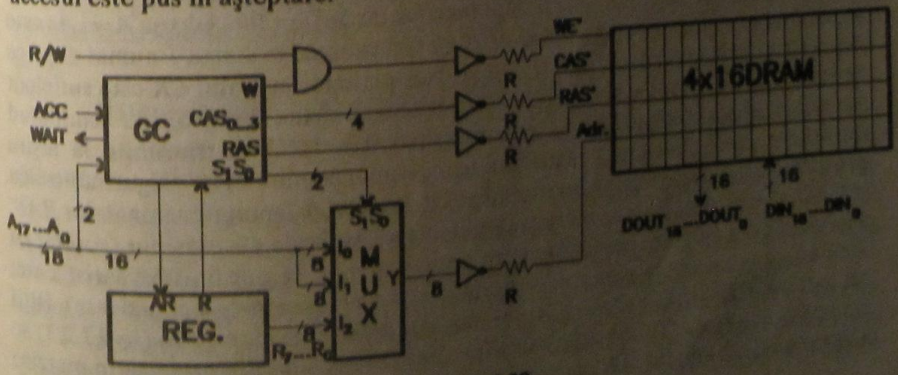



Figura 4.30

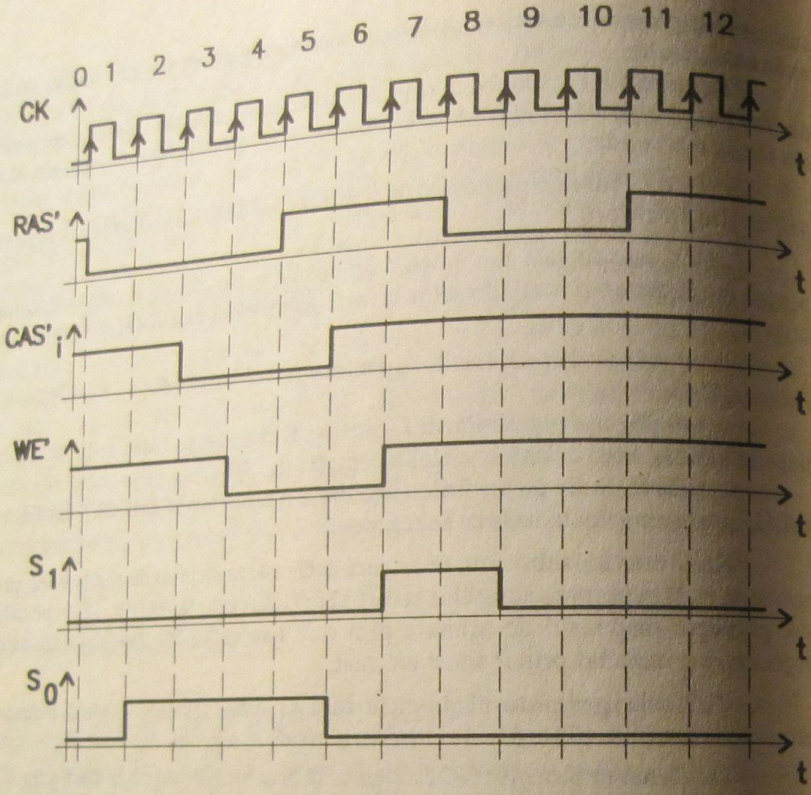



**Figura 4.31**

*GC* este proiectat să funcţioneze ca o ***maşină sincronă*** cu ceasul *CK*, *tranziţia activă* a acestuia fiind cea pozitivă. Deci, orice comutare la ieşirile acestui subsistem este o consecinţă a frontului pozitiv al ceasului. *GC* este inactiv în ciclul *0* al ceasului în care poate rămîne aşteptînd o comandă externă: *ACC* sau *R*. În această *stare* semnalele *RAS'*, *CAS'* şi *WE'* sînt inactive iar $S_1S_0 = 00$. Adresa $A_{17}...A_0$ este setată înaintea activării semnalului *ACC* iar *GC* va trece în starea *1* numai după ce $A_{CC}$ este activat. Se asigură astfel $t_{SURAS}$ dacă perioada ceasului *CK* este suficient de mare. În *1* se garantează $t_{HRAS}$, la sfîrşitul acestui ciclu comută $S_0$ în *1* schimbînd adresa la intrarea cipurilor de memorie deoarece *MUX*-ul transmite la ieşire intrările de tip $I_1$. În ciclul *2* se asigură $t_{SUCAS}$, apoi comută *CAS'* asigurîndu-se cu prisosinţă $t_{HCAS}$ deoarece $S_0$ nu comută decît în ciclul *6*. Duratele semnalelor *RAS'* şi $CAS'_i$ sînt asigurate prin patru şi respectiv trei perioade ale ceasului (corelat cu datele de catelog care, presupunem în acest exemplu, că pot fi astfel satisfăcute; dacă nu, se reconsideră numărul de perioade cît aceste semnale sînt activate). Biţii cei mai semnificativi ai adresei, $A_{17}$ şi $A_{16}$, vor determina care $CAS_i$ $(i=0,1,2,3)$ se activează la ieşirea *GC*. Se asigură astfel rolul *DMUX*-ului din extensia în matrice a *RAM*-ului. Semnalul *WE'* va trebui să prindă ultima parte a *CAS'*-ului, pentru



$t_{SUWE'}$, şi să permită respectarea timpului $t_{HWE'}$. Utilizatorul va asigura stabilitatea adreselor pînă cel puţin în ciclul *3*, inclusiv, şi corelarea datelor la intrările şi ieşirile de date. Semnalul *R/W* indică tipul de ciclu (*R/W = 1* pentru scriere, pentru a genera semnalul *WE'* împreună cu *W* de la ieşirea *GC*).

Dacă se cere ciclu de regenerare (*R=1*) atunci se comută semnalele de selecţie ale *MUX*-ului în $S_1S_0 = 10$ (ciclul *7*) şi urmează un *ciclu de regenerare* în care se activează numai *RAS'*. La ieşirile $R_7 ... R_0$ după fiecare ciclu de regenerare configuraţia binară se *incrementează* (creşte, ca valoare numerică, cu o unitate).

Inversoarele din schemă sînt realizate în varianta de putere (cu curent mare pe ieşire) pentru a asigura comanda unui număr mare de cipuri de memorie. Pe căile de adrese, *RAS'* şi *WE'* sînt conectate *64* de circuite cu capacităţi de intrare de *3 ÷ 10 pF*. Dacă nu dispunem de inversoare suficient de puternice, vom împărţi cipurile în două sau în patru şi le vom comanda cu *drivere* distincte. Rezistenţa *R*, de ordinul zecilor de ohmi, atenuează reflexiile semnalului pe traseele prea lungi impuse de montarea pe placa de cablaj imprimat a unui număr prea mare de cipuri.

Pentru calculul perioadei ceasului va trebui să ţinem cont de timpii de set-up şi hold faţă de *RAS'* şi *CAS'* şi de propagările prin structură. Vom face următoarele notaţii:

- $t_{pRAS}$, $t_{pCAS}$, $t_{pS_1}$, $t_{pS_0}$ - timpii de propagare ai semnalelor *RAS*, *CAS*, $S_1$ şi $S_0$ faţă de frontul activ (pozitiv) al ceasului (nefiguraţi, din considerente de limpezime a reprezentării, în figura 4.31);

- $t_{pS_1Y}$, $t_{pS_0Y}$ - timpii de propagare ai semnalelor de selecţie prin *MUX*;

- $t_{pD}$ - timpul de propagare prin inversoarele folosite ca drivere;

- $t$ - perioada ceasului *CK*.

Folosind notaţiile anterioare vom scrie pentru $t_{SURAS}$

$$t_{SURASmin} = t_{min} - (t_{pSimax} + t_{pSiYmax} + t_{pDmax} - (t_{pRASmin} + t_{pDmin})) .$$

De regulă vom găsi în cataloage timpii maximi de propagare, dar aproape niciodată pe cei minimi. Din acest motiv vom fi obligaţi ca, acoperitor, *timpii minimi să-i considerăm zero pentru o proiectare robustă a schemei*. Relaţia anterioară ne va oferi perioada minimă a ceasului. Considerarea timpilor minimi de propagare ca fiind zero vine să compenzeze efecte neluate în seamă în relaţia anterioară, cum ar fi:

- timpul de întîrziere introdus de conexiuni,

- faptul că pe intrări diferite ale cipului *DRAM* capacitatea parazită este diferită.

O relaţie similară va fi scrisă pentru $t_{HRAS}$ sau $t_{HCAS}$. Vom ţine cont de relaţia care impune condiţiile cele mai restrictive dînd cel mai mare $t_{min}$.

Dezavantajul unui $t_{min}$ mare este evident. Vom lucra cu un modul de timp mare aproximînd grosier ceilalţi timpi impuşi de schemă ($t_{RAS}$ sau $t_{CAS}$).

Atunci cînd aplicaţia cere un timing foarte strîns o astfel de soluţie nu este posibilă. Vor fi folosite alte tehnici ce fac apel la linii de întîrziere de precizie cu care se vor putea impune relaţii temporale oarecare nu restricţionate la un multiplu

de *t*, ce uneori poate creşte cu pînă la *50%* durata ciclului faţă de ciclul minim admis de foaia de catalog.

Proiectarea în detaliu a *GC* va putea fi urmărită într-unul din capitolele următoare. □

## 4.4 Structuri master-slave

### 4.4.1 Principiul master-slave

Introducerea latch-ului cu ceas a realizat numai o decuplare parţială între *cînd?* şi *cum?* în comutarea unui circuit basculant bistabil. Am văzut că pe durata palierului activ al ceasului bistabilul era transparent la modificările apărute pe intrări. Latch-ul cu ceas comută comandat de *palierul activ al ceasului* şi dorim să construim un circuit de memorare bistabil care să comute sincronizat cu *frontul activ al ceasului*. Se impune restrîngerea preciziei în determinarea momentului comutării la momentul tranziţiei frontului unui impuls. Va trebui să putem spune că bistabilul comută într-un moment comandat de frontul ceasului şi numai de acesta; adică să realizăm o decuplare maximă între *cînd?* şi *cum?*.

• **Circuitul master-slave** într-o variantă foarte simplă este prezentat în figura 4.32*a*. El este format din două latch-uri cu ceas, conectate în cascadă şi comandate cu ceasul în antifază. Acest circuit reprezintă astfel *extensia* minimă *serie în clasa sistemelor de ordinul unu*. Pentru *CK=1* este transparent primul latch, *master-ul (M)*, iar cel de al doilea menţine o stare anterior comandată. Cînd *CK* comută din *1* în *0* este închis primul latch şi devine transparent cel de al doilea, *slave-ul (S)*, care copiază conţinutul master-ului, aşa cum a fost el stabilit prin jocul intrărilor sale înainte de tranziţia negativă a ceasului. Deci, ieşirea circuitului va comuta drept consecinţă a tranziţiei negative a *CK*-ului, conform configuraţiei stabilite pe intrări imediat înainte de această tranziţie.

Momentul tranziţiei este numai aparent un moment deoarece decuplarea între *cînd?* şi *cum?* nu este perfectă nici în acest caz. Intrările circuitului nu se pot modifica în orice moment. Este definit şi în jurul frontului activ un interval de timp în care intrările trebuie să fie stabile (figura 4.32*c*). Este vorba de timpii de set-up şi hold faţă de frontul activ al ceasului. În intervalul $t_{SU} + t_H$ ce include tranziţia activă a *CK* intrările trebuie menţinute stabile. Rezultă că, de fapt, circuitul master-slave *aproximează momentul cu un interval*. Cînd spunem că în bistabil se încarcă o stare conformă cu intrările la *momentul* tranziţiei frontului, spunem de fapt că încărcăm în bistabil o stare conformă intrării menţinute stabile *un interval de timp* înainte ($t_{SU}$) şi un interval de timp după ($t_H$) frontul activ. Din punct de vedere ingineresc momentul nu poate exista, întotdeauna va fi aproximat printr-un interval.



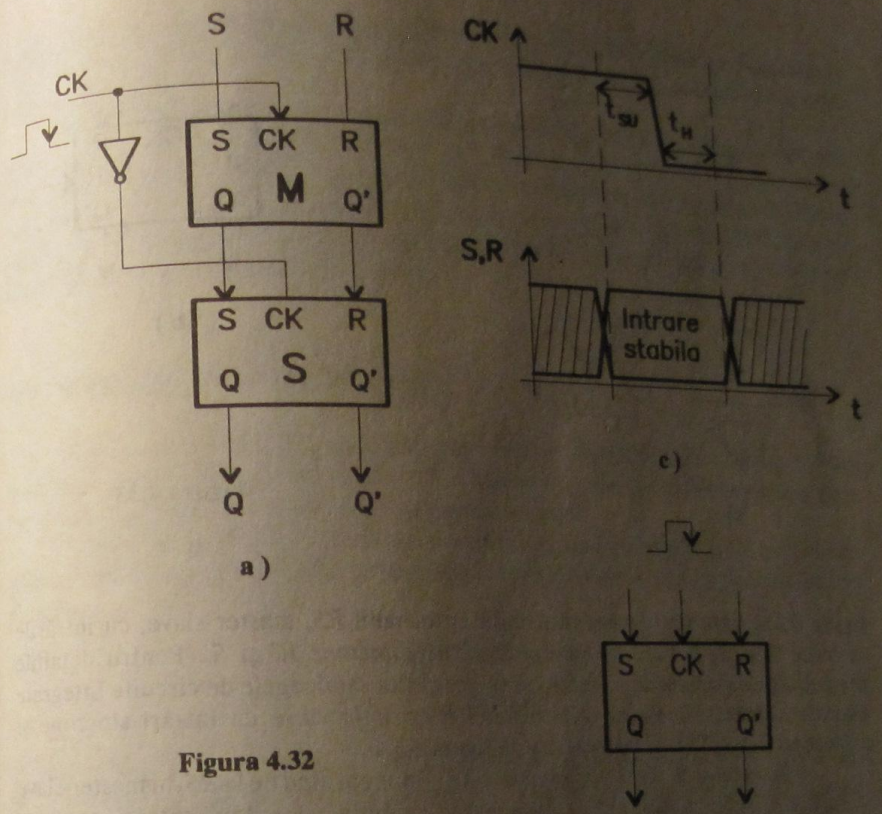



**Figura 4.32**

Comutarea unui bistabil va mai fi caracterizată de timpii de propagare:

- $t_{pLH}$ - timpul de propagare la comutarea ieşirii din *0* în *1* măsurat ca întîrziere faţă de frontul activ al ceasului;

- $t_{pHL}$ - timpul de propagare de la *CK* la *Q* sau *Q'* cînd ieşirea comută din *1* în *0*.

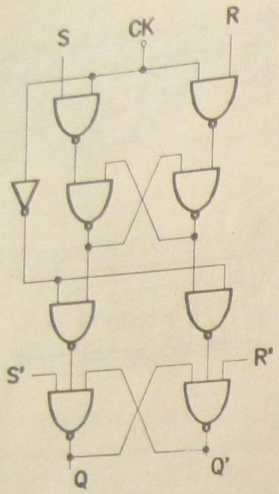
De regulă $t_{pLH} \neq t_{pHL}$ ca şi în cazul circuitelor combinaţionale.

• O inegalitate foarte importantă este următoarea:

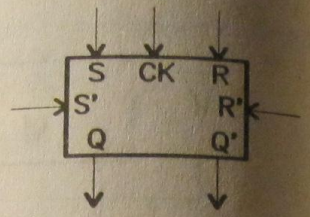
**Propoziţia 4.2** $t_H < min\ (t_{pLH}, t_{pHL})$ . □

În construcţia sistemelor de ordinul doi va fi esenţială. Analiza sumară a shemei oricărei structuri master-slave va permite cititorului demonstrarea inegalităţii anterioare.

• **Intrările de tip set** (S') **şi reset** (R') vor fi regăsite şi la bistabilii de tip master-slave. Aceste intrări acţionează la nivelul slave-ului, necondiţionate de ceas. Ele se aplică direct latch-ului elementar de la acest nivel, deoarece porţile cu care acesta este realizat posedă cîte trei intrări, una pentru bucla locală, alta pentru ieşirile masterului condiţionate de ceas şi una pentru *S'* sau *R'* aplicate direct. În




a)




b)

**Figura 4.33**

figura 4.33*a* vom reprezenta în detaliu un bistabil *RS*, master slave, cu intrările *sincrone* *R* şi *S*, la care se adaugă intrările *asincrone* *R'* şi *S'*. Pentru detaliile prezentate de structurile concrete se pot consulta cataloagele de circuite integrate digitale. Simbolul logic al unui bistabil *RS* master slave cu intrări sincrone şi asincrone (*S'* şi *R'*) este reprezentat în figura 4.33*b*.

• **Bistabilul de tip D** (delay) poate fi obţinut pornind de la *RS*-ul master-slave (figura 4.32*b*) restricţionînd cu un inversor evoluţia celor două intrări, conform reprezentării din figura 4.34*a*. Rezultă circuitul, reprezentat schematic în figura

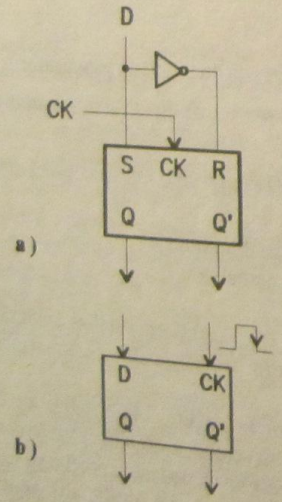



**Figura 4.34**

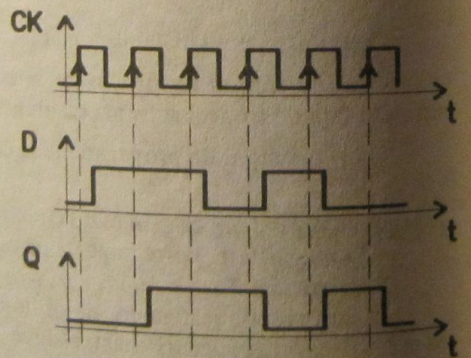



**Figura 4.35**



4.34*b*, care se deosebeşte esenţial de latch-ul de tip *D* (*data*) care este un circuit *transparent pe palierul activ al ceasului*, spre deosebire de cel acum obţinut care *comută pe front*.

• **Bistabilul de tip D funcţionează ca circuit de întîrziere** (delay) deoarece generează în ciclul *n* al ceasului ceea ce a avut pe intrare în ciclul anterior, al (*n-1*)-lea. Acest circuit realizează astfel o *întîrziere secvenţială*, conform reprezentării din figura 4.35, unde intrarea *D* evoluează condiţionat tot de frontul pozitiv al ceasului într-un sistem oarecare. Se observă că ieşirea *Q* apare decalată (întîrziată) cu un tact faţă de intrare.

## 4.4.2 Registrul de deplasare serie

Una din aplicaţiile cele mai importante ale bistabilului de tip *D* este realizarea *registrelor de deplasare serie* bazate tocmai pe capacitatea lor de a funcţiona ca circuite de întîrziere comandate de ceas.

Un astfel de registru se obţine conectînd în cascadă bistabile de tip *D* ce sînt comandate de acelaşi impuls de tact (figura 4.36). Drept urmare a fiecărui impuls de ceas

$$Q_1 \leftarrow DIN,$$

$$Q_i \leftarrow Q_{i-1} \quad \text{pentru } i>1,$$

iar valoarea lui $Q_n$ se pierde. Conform *Propoziţiei 4.2* o astfel de tranziţie este posibilă dacă circuitele sînt de acelaşi tip, în sensul că, sînt realizate în aceeaşi tehnologie, la parametrii asemănători. Altfel, conectarea în cascadă este imposibilă pentru că intrările unor bistabili s-ar putea modifica în intervalul $t_H$, timpul de propagare al circuitului anterior fiind foarte mic.

Dacă circuitele ar fi de tip latch cu ceas, atunci, în intervalul de transparenţă, semnalul s-ar propaga printr-un număr nedefinit de circuite, funcţie de durata palierului activ. Am avea de-a face cu un proces necontrolat şi, în consecinţă, inutilizabil într-o aplicaţie oarecare.

Pentru a înţelege mai bine procesul de propagare putem folosi şi o altă reprezentare (figura 4.37) care o detaliază pe prima pînă la nivelul latch-urilor cu ceas de tip *D* (*data*). În acest circuit propagarea se produce pe un traseu de forma unor dinţi de fierăstrău. Pentru *CK=1* sînt transparente latch-urile superioare iar

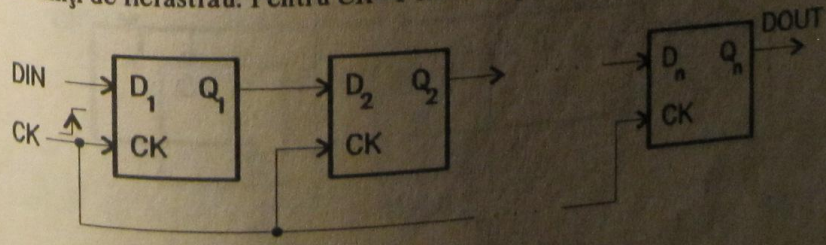



**Figura 4.36**

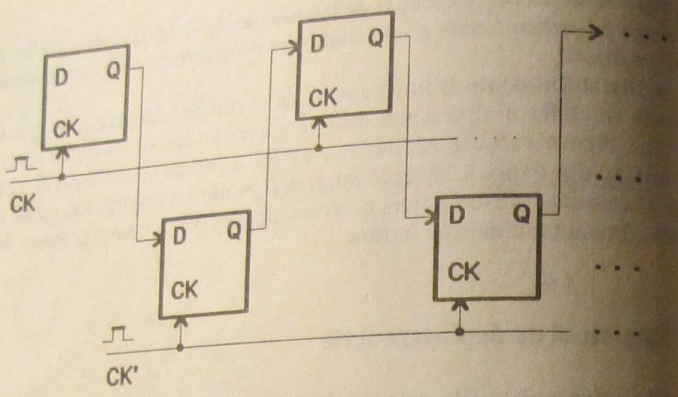



Figura 4.37

pentru *CK=0* cele inferioare. În primul caz se încarcă bistabilii de sus iar apoi, la comutarea în zero a ceasului, conţinutul lor se varsă în cei de jos. La fiecare ceas datele avansează cu "un dinte". Evident, două latch-uri succesive formează un bistabil master-slave de tip *D* (*d*elay).

În cazul registrelor de foarte mare dimensiune se pune şi problema sincronizării ceasului la toate intrările. Într-o structură geografic distribuită asigurarea ceasului sincron este o problemă, datorită întîrzierilor pe conexiuni şi din cauza driverelor suplimentare impuse de fan-out-ul limitat.

Un bit introdus pe *DIN* se regăseşte pe *DOUT* după un număr de *n* tacturi. Spunem că o configuraţie binară introdusă bit cu bit pe *DIN* se regăseşte *întîrziată cu n tacturi* la ieşirea *DOUT*. Se pot imagina şi "prize" după un număr oarecare de celule, mai mic ca *n*, astfel încît să avem întîrzieri de valori diferite. În cazul general se poate imagina schema din figura 4.38 care asigură orice întîrziere între *0* şi *n* tacturi, comandat pe codul de selecţie al *MUX*-ului.

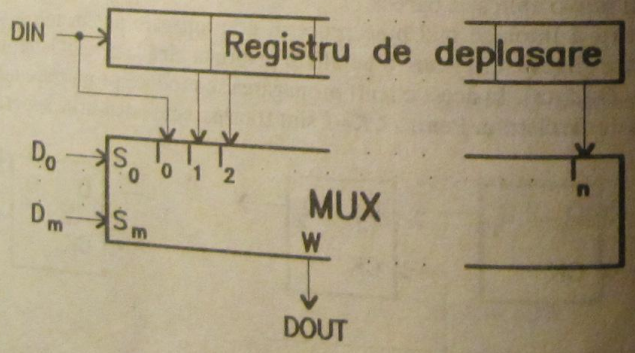



Figura 4.38



## 4.5 Registre

### 4.5.1 Registrul paralel

*Extensia serie-paralel* în clasa sistemelor de ordinul unu o reprezintă *registrul paralel*. Acest dispozitiv se obţine *conectînd în paralel bistabile master-slave de tip D* ce sînt extensii serie în sistemele de ordinul unu. Termenul de registru este folosit corect numai în cazul acestor structuri. În alte cazuri avem de a face cu extensii forţate ale sensului de registru (spre exemplu: registru de deplasare serie).

Dacă într-un bistabil înscriem, sincron cu un front, un bit (cu valoarea *0* sau *1*), într-un registru de *n* biţi înscriem, tot pe front, o configuraţie binară de *n* biţi. Registrul va permite, de asemenea, realizarea întîrzierii unei configuraţii binare cu un tact.

### 4.5.2 Registrul serie - paralel

O combinaţie curentă o reprezintă *registrul serie-paralel* ce îmbină în aceeaşi structură funcţia de *registru* cu cea de *registru de deplasare serie*. Schema de principiu este reprezentată în figura 4.39, unde fiecare celulă este formată dintr-un bistabil de tip *D* şi un *MUX* de două căi, una pentru încărcarea ca registru (paralel) şi alta pentru deplasarea serie.

Dacă semnalul *M* (mod) este pe zero, atunci are loc deplasarea serie, în fiecare bistabil, înscriindu-se valoarea din cel anterior, în primul încărcîndu-se *SI* (*s*erial *i*nput), iar valoarea din ultimul transmiţîndu-se în exterior pe calea *SO* (*s*erial *o*utput).

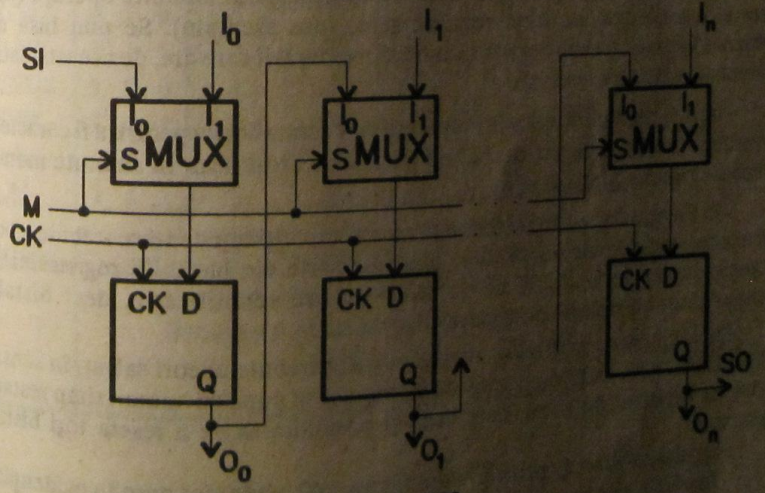



Figura 4.39

Structura generală mai poate fi simplificată, în majoritatea cazurilor din considerente de încapsulare a structurii într-o capsulă cu un număr limitat de conexiuni. Astfel pot apare:

- registru cu intrare serie şi ieşire paralel,

- registru cu intrare paralel şi ieşire serie.

Varianta completă este disponibilă numai pentru un număr redus de celule (exemplu: 7495).

Pentru detalii se pot consulta cataloagele firmelor producătoare. Înainte de a începe orice proiect este bine să ne familiarizăm cu facilităţile la funcţionare de care dispune fiecare circuit în parte. Circuitele din seriile standard au particularităţi utile care, odată cunoscute, oferă sugestii utile proiectantului. Deci, cunoaşterea restricţiilor şi facilităţilor oferite de structurile concrete, comercializate, este esenţială în precizarea opţiunilor proiectantului. Lecturarea curentă a cataloagelor este o îndeletnicire obligatorie a oricărui inginer proiectant.

## 4.6 Registrul de stare

Descrierea funcţiilor unei unităţi logico-aritmetice (*ALU*) a pus în evidenţă ieşiri ce caracterizau rezultatul operaţiei efectuate. De regulă, rezultatele unei operaţii logico-aritmetice sînt stocate, în registre de date sau locaţii de memorie, pentru o folosire ulterioară (după cum se va vedea în continuare). Biţii ce caracterizează rezultatul operaţiei (spre exemplu: *CR*, *ZERO*, *A=B* ş.a.) vor fi stocaţi în *registrul de stare* pentru a putea fi ulterior testaţi în vederea unor eventuale decizii bazate pe modul în care s-au desfăşurat anumite operaţii (dacă o operaţie a avut sau nu drept rezultat zero, spre exemplu). Se pun însă unele probleme în cazul acestui registru, datorită memorării care are, de această dată, un caracter selectiv, în sensul că:

- numai rezultatul anumitor operaţii este semnificativ pentru a fi caracterizat prin starea acestui registru, deci el nu trebuie încărcat decît în anumite momente ce trebuie riguros controlate;

- în funcţie de operaţia efectuată numai anumiţi biţi de stare trebuie afectaţi, deci comutarea trebuie să afecteze numai o parte din bistabilii registrului (la o operaţie logică ieşirea *CR* de la *ALU* nu are semnificaţie, deci bistabilul corespunzător nu trebuie să comute).

De asemenea, conţinutul acestui registru trebuie uneori salvat, în sensul că trebuie stocat într-un alt dispozitiv de memorare, şi după un anumit timp restaurat, readus în registrul de stare. Este utilă şi posibilitatea de a reseta toţi bistabilii registrului.

Rezultă schema de principiu din figura 4.40, unde sînt puse în evidenţă:

- Intrările pe care se aplică indicatorii,



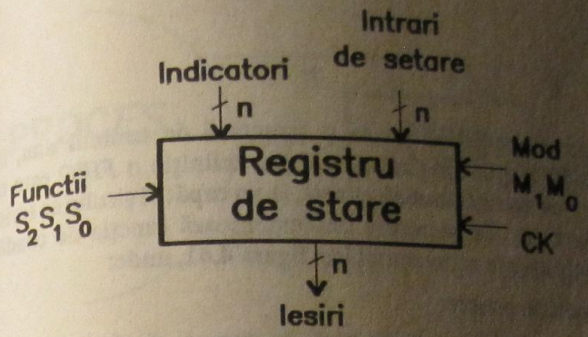



Figura 4.40

- Ieşirile bistabililor de stare din registru,

- Intrările de setare pentru restaurarea conţinutului registrului,

- Intrările $S_2S_1S_0$ ce reprezintă codurile funcţiilor efectuate în *ALU* (în acest caz este vorba de un *ALU* cu opt funcţii numai),

- Intrările $M_1M_0$, pe care specificăm modul de lucru al registrului în ciclul curent de ceas,

- Intrarea de ceas, *CK*.

Codurile funcţiilor efectuate în *ALU* sînt necesare la intrarea acestei structuri pentru a determina care sînt biţii de stare afectaţi în ciclul curent. În tabelul 4.1 este definită corelaţia dintre funcţia logico- aritmetică şi indicatorii de stare. Modul de funcţionare al registrului de stare este selectat conform codurilor din tabelul 4.2.

Lăsăm cititorului, ca exerciţiu, plăcerea proiectării acestei structuri.

| Funcţie | $S_2$ $S_1$ $S_0$ | "a=b" | "zero" | " cr " | " par " | " sgn " |
|---|---|---|---|---|---|---|
| AND | 0 0 0 | 1 | 1 | 0 | 0 | 0 |
| OR | 0 0 1 | 1 | 1 | 0 | 0 | 0 |
| XOR | 0 1 0 | 1 | 1 | 0 | 0 | 0 |
| PLUS | 0 1 1 | 1 | 1 | 1 | 1 | 1 |
| MINUS | 1 0 0 | 1 | 1 | 1 | 1 | 1 |
| SHR | 1 0 1 | 0 | 1 | 0 | 1 | 0 |
| A | 1 1 0 | 0 | 1 | 0 | 1 | 1 |
| A' | 1 1 1 | 0 | 1 | 0 | 1 | 1 |

Tabelul 4.1

| $M_1$ $M_0$ | Mod de lucru |
|---|---|
| 0 0 | Actualizează indicatori de stare |
| 0 1 | Reîncarcă indicatorii de stare |
| 1 0 | Reset |
| 1 1 | Noop |

Tabelul 4.2

## 4.7 Memoria FIFO

Un alt exemplu, care poate fi implementat ca o structură de ordinul unu, îl constituie memoria de tip *FIFO* (*first-in-first-out*). Prin definiţie o *FIFO* este o memorie structurată ca *un şir cu acces la ambele capete*; la un capăt al şirului se scrie iar la celălalt se citeşte. Acest tip de memorie implementează funcţia de coadă (queue). Schema sa de principiu este reprezentată în figura 4.41, unde:

- *W* reprezintă comanda de scriere;

- *R* reprezintă comanda de citire distructivă, în sensul că simbolul citit dispare din şirul memorat făcînd loc unei eventuale noi scrieri în structura finită a şirului;

- *EMPTY* este un semnal ce indică şirul vid (memoria nu conţine nici un simbol);

- *FULL* este semnalul ce indică faptul că memoria este plină; nu mai poate fi făcută nici o scriere înainte de a se efectua cel puţin o citire;

- *DIN* şi *DOUT* reprezintă intrarea şi respectiv ieşirea de date;

- *CK* este intrarea de ceas.

• Structura concretă a unei memorii *FIFO* va fi exemplificată în capitolul următor, sub forma unei maşini de ordinul doi ce permite o realizare mai eficientă şi mai fiabilă. Cititorul îşi poate imagina singur o *FIFO* ca *SO1*, sesizînd cu această ocazie şi limitele unei astfel de implementări.

• O aplicaţie curentă a memoriei *FIFO* este cea de *memorie tampon* între două procese. În figura 4.42 sînt reprezentate schematic două procese, *PROCES 1* şi *PROCES 2*, interconectate prin intermediul unei astfel de memorii. Desemnăm prin proces mai mult decît o simplă structură fizică. Se poate vorbi de un proces de calcul oarecare, generalizînd structura *FIFO* la una abstractă de tampon între două structuri fizice sau două procese abstracte.

Ideea de bază este că *PROCES 1* şi *PROCES 2* sînt *asincrone*, unul generînd şi celălalt recepţionînd date, cu *rate medii apropiate*. Astfel, un anumit interval de

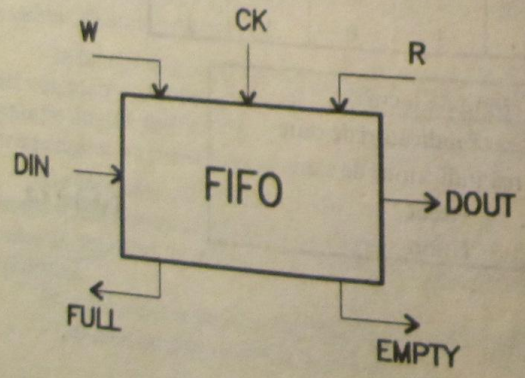



**Figura 4.41**



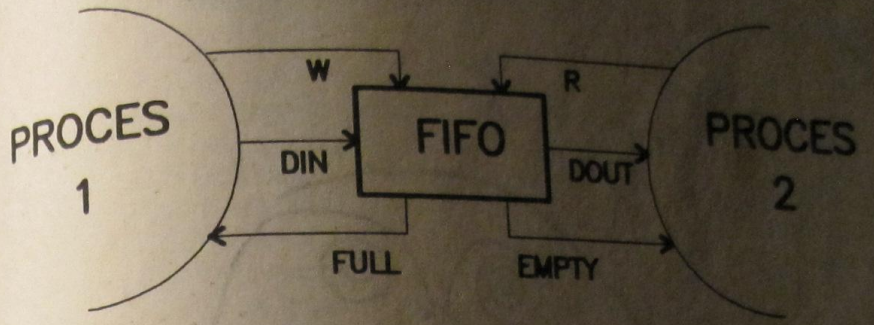
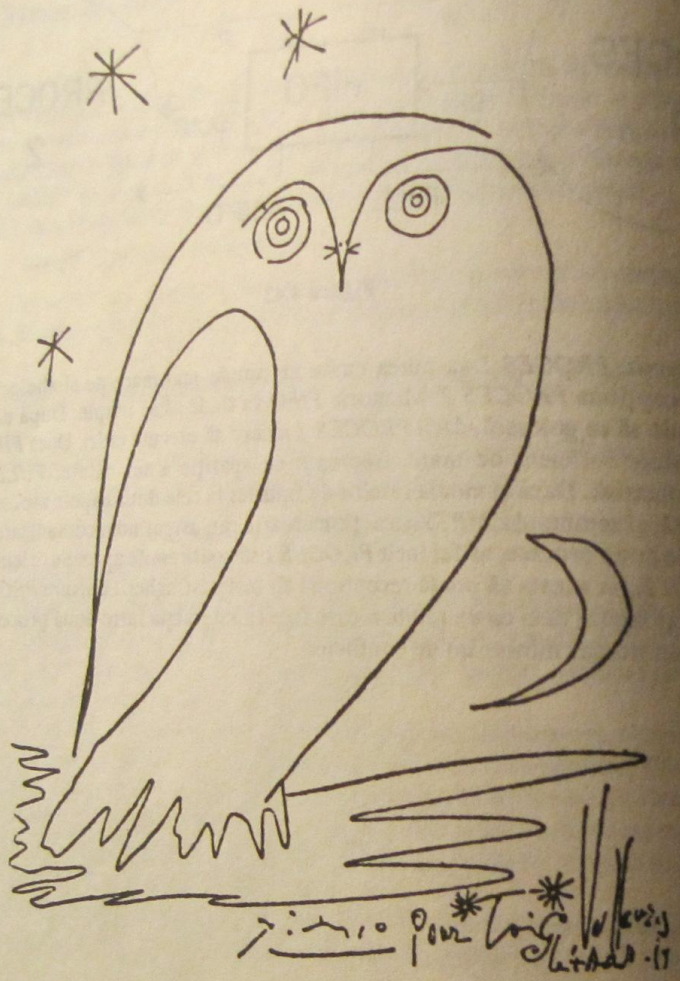



**Figura 4.42**

timp, limitat, *PROCES 1* va putea emite un număr mai mare de simboluri decît poate recepţiona *PROCES 2*. Memoria *FIFO* va tinde să se umple. După un timp ea va tinde să se golească, dacă *PROCES 2* începe să execute citiri. Dacă *FIFO* are o capacitate suficient de mare, frecvenţa de apariţie a semnalului *FULL* va fi nesemnificativă. Dacă şi media ratelor de transfer la cele două capete ale *cozii* este apropiată, şi semnalul *EMPTY* va acţiona foarte rar, asigurînd o comunicare lejeră între cele două procese, astfel încît *PROCES 1* să poată emite aproape oricînd către *PROCES 2*, iar acesta să poată recepţiona cît mai nestingherit dinspre *PROCES 1*. *FIFO* acţionează deci ca un *tampon* care face ca interacţia între două procese să se facă cu un număr minimizat de conflicte.



P. Picasso : *Desen.*

5

# AUTOMATE FINITE ELEMENTARE
## Sisteme de ordinul doi

Prima buclă închisă peste un circuit combinaţional a conferit structurii rezultate posibilitatea unei *stări autonome*, parţial independente de evoluţia intrării. Această rudimentară autonomie va fi sporită prin cea de a doua buclă de reacţie prin care obţinem *sistemele de ordinul doi* (*SO2*). La acest nivel sistemele digitale dobîndesc *autonomie în comportament*, în sensul unei *evoluţii* a ieşirii parţial sau total independentă de evoluţia intrării. Fiecare nouă buclă a adus şi va aduce, în continuare, un spor al autonomiei structurilor digitale.

Din punct de vedere funcţional aceste circuite sînt orientate către

- sisteme de comandă şi control,
- sisteme de prelucrare secvenţială,
- sisteme de generare a şirurilor de simboluri.

Structural, deosebirile între aceste trei categorii de aplicaţii sînt neesenţiale, motiv pentru care tratarea ce urmează nu se va face din această perspectivă funcţională, dar de fiecare dată, cînd va fi cazul, vor apare comentarii referitor la domeniul de utilizare.

## 5.1 Structura de principiu a unui automat finit

Un automat finit se va structura prin introducerea unei bucle suplimentare într-un sistem de ordinul unu. Acest proces poate genera cele mai diverse configuraţii de circuit. Din punct de vedere tehnic numai o parte vor putea fi luate în consideraţie, ţinînd cont de restricţii tehnologice de proiectare, realizare, testare şi întreţinere. Evoluţia tehnologiilor determină, desigur, reorientări uneori spectaculoase. În acest capitol vom aborda automatele finite din perspectiva restricţiilor tehnologice şi conceptuale curente. În urmă cu 20 de ani un astfel de subiect ar fi fost, cu siguranţă, altfel tratat, iar peste alţi 20 va arăta cu totul schimbat, cu toate că fundamentarea teoretică nu a suportat şi nu va suporta mutaţii spectaculoase. O oarecare tendinţă de evoluţie vom pune în evidenţă chiar în acest text (este vorba de tendinţa de structurare echilibrată ce tinde să se impună).

### 5.1.1 Segregarea funcţională

Structura de principiu a unui *automat finit elementar* este prezentată în figura 5.1, unde, peste un sistem de ordinul unu a mai fost închisă o buclă de reacţie, conferind, prin aceasta, ansamblului ordinul doi. În reţeaua iniţială există circuite combinaţionale şi circuite de memorare interconectate, astfel încît să poată executa o anumită funcţie.

**Exemplul 5.1** Circuitul din figura 5.2 este un automat finit. Dacă îi analizăm structura, pot fi puse în evidenţă două niveluri la care se închid bucle de reacţie. Răspunsul obţinut pe ieşirile $Q_1$ şi $Q_0$ depinde, după cum se poate constata, atît de intrările *A, B, C* şi *D* cît şi de ieşirile înseşi. Spre exemplu, dacă $Q_1=Q_0=0$, atunci orice modificare a valorilor variabilelor de intrare *C* şi *D* nu este sesizabilă la ieşire. Dar, dacă $Q_1Q_0 = 01$, atunci ieşirea $Q_1$ va fi sensibilă la comutarea din *0* în *1* a variabilei *C*. Pentru $Q_1Q_0 = 10$ ieşirea este complet insensibilă la comutarea variabilelor de intrare.

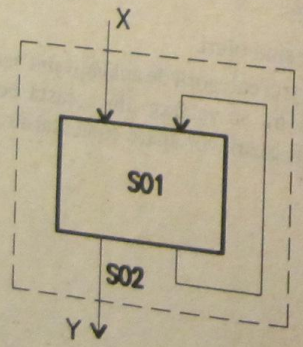



**Figura 5.1**



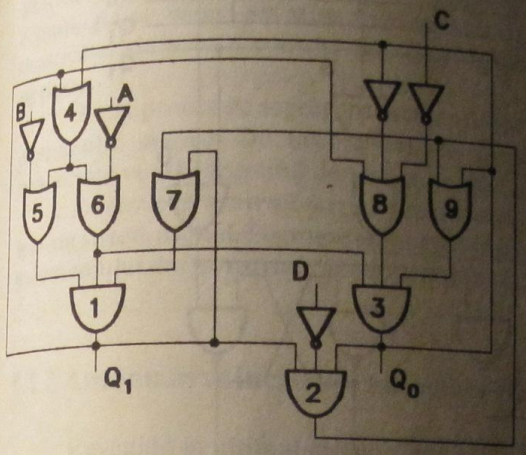



**Figura 5.2**

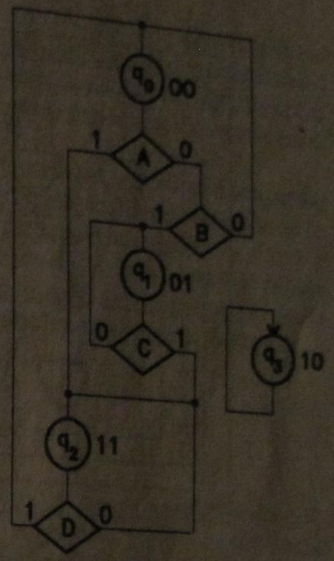



**Figura 5.3**

În urma unei analize atente vom putea descrie comportamentul circuitului prin organigrama din figura 5.3. Comutarea ieşirilor este determinată strict de comutarea variabilelor de intrare. Configuraţiile binare de *ieşire* $q_i$ ($i = 0, 1, 2, 3$), codificate prin doi biţi, $Q_1$ şi $Q_0$, reprezintă, în acelaşi timp şi *starea* circuitului.

O analiză atentă a circuitului pune în evidenţă structuri elementare de memorare sub forma latch-urilor realizate cu circuitele *1, 7* şi *3, 9* (conform principiului reprezentat în figura 4.6*d*). Circuitele *AND 1* şi *3* contribuie, în acelaşi timp, şi la realizarea unor funcţii logice. Nu se poate separa net, în această structură, zona elementelor de memorare de cea a circuitelor de prelucrare logică.

Operînd transformări logice elementare se poate reconfigura circuitul sub forma celui din figura 5.4, în care sînt puse net în evidenţă două zone:

- zona circuitelor elementare de memorare (cele două latch-uri de sub linia punctată);

- zona circuitelor combinaţionale de prelucrare, deasupra liniei punctate.

Sistemele de ordinul zero sînt astfel net separate de cele de ordinul unu. În noua formă este perfect conservată funcţia. □

Exemplul anterior pune în evidenţă *posibilitatea segregării funcţionale* într-un automat finit. Este vorba de segregarea dintre elementele de *memorare* (*SO1*) şi cele de *prelucrare* (*SO0*). Acest proces de segregare nu este obligatoriu dar este deosebit de util pentru că distinge între două subsisteme ce au legi de creştere net distincte:

- subsistemul de memorare, care poate fi definit foarte simplu şi are o evoluţie (creştere) cantitativă liniară (dacă vrem să crească cu un bit, se adaugă, simplu, o celulă ce memorează un bit);

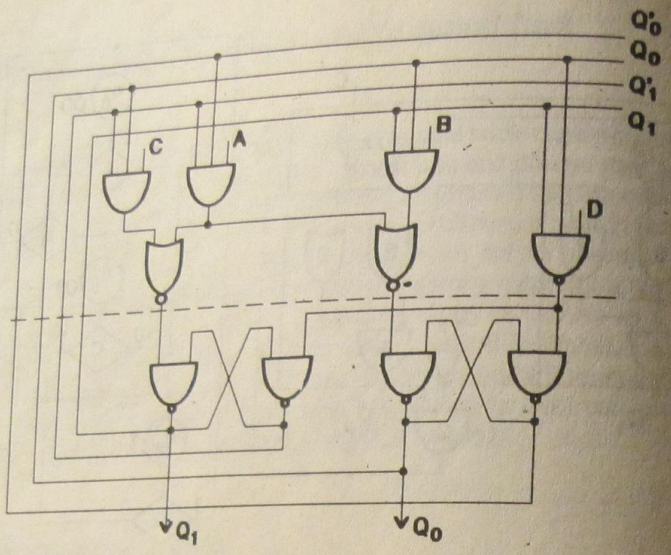



**Figura 5.4**

- subsistemul de prelucrare cu o complexitate mai mare şi manieră de definire mai dificilă.

În proiectarea unui automat efortul este concentrat asupra zonei de circuit combinaţional, memoria fiind alocată, simplu, conform numărului de biţi impus de definiţie. Segregarea ne ajută tocmai prin posibilitatea de a ne putea concentra asupra părţii de circuit care depinde de funcţia concretă asociată automatului. Într-adevăr, *automatul este concretizat prin CLC-ul asociat*, odată ce a fost stabilită dimensiunea memoriei asociate.

*Din considerente de eficienţă a proiectării, realizării, testării şi întreţinerii, automatele finite vor fi gîndite în varianta segregată funcţional* prezentată în figura 5.5 (unde nu sînt specificate ieşirile automatului). Menţionăm faptul că această

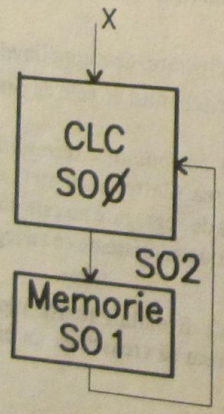



**Figura 5.5**



opţiune este numai inginereşte motivată, din punct de vedere teoretic nefiind absolut necesară. Motivaţia fiind ingineracească, este condiţionată de nivelul tehnologic pe care l-au atins implementările concrete ale automatelor finite. O drastică schimbare a contextului tehnologic ar putea repune în discuţie această opţiune.

Acest proces de segregare funcţională se va mai declanşa şi la alte paliere ale demersului nostru. De fiecare dată va fi motivat de opţiuni pragmatice şi în consecinţă va fi susceptibil de revizuiri odată cu evoluţia tehnologiilor.

*Prelucrarea* şi *memorarea* se conturează în acest moment ca funcţii esenţiale într-un sistem digital. Segregarea lor în continuare asigură o evoluţie spectaculoasă a domeniului dar constituie şi germenele limitărilor fundamentale ce vor apare.

## 5.1.2 Automate asincrone - automate sincrone

Revenind la ultimul exemplu remarcăm faptul că tranziţia automatului este dată de comutarea variabilelor de intrare care determină, parţial, *cum* comută automatul şi, total, *cînd* comută automatul. Această nedecuplare între *cînd* şi *cum* a fost sesizată ca supărătoare şi în cazul structurilor elementare de memorare şi a fost rezolvată prin introducerea *semnalului de tact* (ceas), care *sincroniza* funcţionarea, comutarea, latch-urilor. Automatul anterior prezentat este un *automat asincron* deoarece comutarea sa nu este sincronizată prin impulsuri de ceas. Tranziţia unui astfel de circuit este la discreţia comutării variabilelor de intrare.

• **Pentru a realiza decuplarea între *cînd* şi *cum* comută automatele, s-a impus conceptul de *automat sincron*.** Decuplarea este efectiv realizabilă numai într-o structură segregată funcţional deoarece

- problema lui *cum* se rezolvă la nivelul circuitelor de *prelucrare* combinaţională,

- problema lui *cînd* implică elemente de *memorare* sincronizate cu semnale de tact.

Deci, *o primă completare* a schemei din figura 5.5 s-ar putea realiza aplicînd semnalul de ceas (*CK*) circuitului de memorare care se restructurează corespunzător.

• *A doua observaţie* ce poate fi făcută porneşte de la următoarea situaţie:

- automatul se află în starea $Q_1Q_0 = 00$ cu $A=B=C=D=0$,

- intrările *B* şi *C* comută în *1* pentru un interval de timp egal cu *t*.

Conform organigramei din figura 5.3 automatul trece în starea $Q_1Q_0=11$, trecînd prin starea $Q_1Q_0=01$: În starea intermediară automatul rămîne un interval de timp ce ţine de timpii de propagare asociaţi circuitelor elementare din schemă. Dacă *t* este prea mic s-ar putea ca automatul să nu ajungă decît în starea $Q_1Q_0=01$. Funcţionarea automatului depinde astfel de durata comenzilor aplicate intrării.

Ne putem pune întrebarea dacă starea intermediară este sesizată de circuitele comandate de ieşirile $Q_1Q_0$. Acest timp depinzînd de propagarea prin porţi, al căror timp minim de propagare nu este, de regulă, garantat, nu va putea fi niciodată evaluat corect. În bună tradiţie inginerească va trebui să-l considerăm nul.

Deci, automatul, evoluînd numai sub controlul variabilelor de intrare, poate avea evoluţii neriguros definite.

Lăsăm cititorului plăcerea de a-şi imagina ce se petrece cu automatul care, pornind din aceeaşi situaţie iniţială, evoluează sub comanda $A=D=1$ şi $B=C=0$ un interval de timp $t$.

• *O altă limitare* ce apare, în cazul exemplului de automat dat, este aceea că evoluţia, comutarea, ieşirii se produce numai strict condiţionat de comutarea intrărilor. Un comportament autonom ar presupune şi o *evoluţie a ieşirii independentă de comutarea intrărilor*. Acest lucru am văzut că este posibil dar nu într-o manieră controlată. Va trebui găsit un procedeu ca, indiferent de cum comută configuraţiile binare aplicate intrării automatului, durata fiecărei stări să poată fi controlată, să fie, eventual, aceeaşi. Dacă cititorul a urmat îndemnul din alineatul anterior, a putut constata că automatul poate avea un comportament pe ieşire, în condiţii de menţinere a configuraţiei binare pe intrare, dar dimensionarea timpului nu poate fi ţinută sub control. Starea $Q_1Q_0=00$ este urmată de starea $Q_1Q_0=11$ la un interval de timp necontrolabil. La fel starea *11* este urmată de *00* la fel de netemperat. Acest lucru este datorat faptului că *orice modificare la ieşire este imediat resimţită pe bucla automatului determinînd o nouă modificare a ieşirii*. Cu alte cuvinte *bucla de ordinul doi se închide continuu* generînd un comportament a căruĭ desfăşurare în timp depinde numai de întîrzierile asociate acestei bucle.

Pentru a putea controla secvenţarea stărilor va trebui să avem pe bucla automatului un *element de memorie netransparent*, care nu permite închiderea continuă a buclei. Deci, memoria pe buclă va fi realizată cu *bistabile de tip master-slave* ce comută comandate de *frontul* impulsului de ceas. Astfel, bucla nu se va mai închide decît ca o consecinţă a frontului activ al tactului, automatul realizînd, cu această ocazie, numai tranziţia în starea strict următoare la fiecare impuls de ceas. Fiecare perioadă a ceasului va avea asociată o stare a automatului şi numai una.

**Exemplul 5.2** Pornind de la organigrama din figura 5.3 vom construi circuitul asociat. Ca element de memorie vom folosi un registru de doi biţi realizat cu două bistabile master-slave de tip *D*, conform reprezentării din figura 5.6. Inspectînd organigrama observăm că $Q_1$ trebuie să comute în *1* cu semnalul de ceas, ori de cîte ori $Q_1=Q_0=0$ şi $A=1$ sau $Q_1=0$ şi $Q_0=1$ şi $C=1$ sau $Q_1=Q_0=1$ şi $D=0$, ceea ce revine la a scrie:

$$D_1 = Q'_1Q'_0A + Q'_1Q_0C + Q_1Q_0D' .$$

Similar scriem

$$D_0 = Q'_1Q'_0A + Q'_1Q_0C + Q_1Q_0D' + Q'_1Q'_0A'B =$$

$$= D_1 + Q'_1Q'_0B .$$



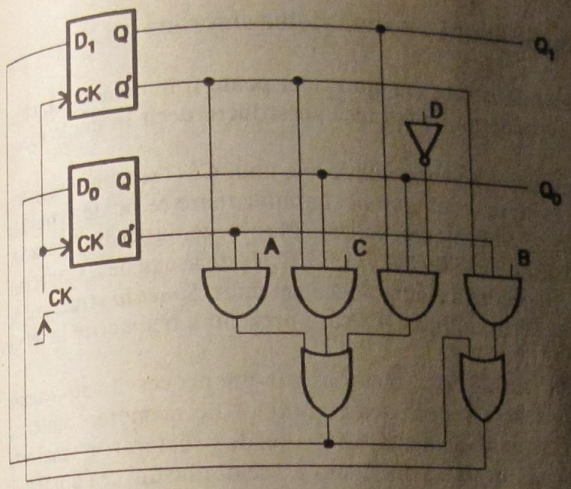



Figura 5.6

Rezultă circuitul din figura 5.6. (Din starea $Q_1Q_0=10$ automatul va trece în starea $Q_1Q_0=00$ pentru a-şi amorsa funcţionarea.)

Dacă perioada ceasului este mai mare decît timpul maxim de propagare pe buclă, atunci automatul va funcţiona corect. Adică

$$T \geq t_{pD} + t_{pAND} + 2t_{pOR} + t_{SU} .$$

(Evident va fi îndeplinită condiţia ca

$$t_H < T_{min} - t_{SU},$$

conform *Propoziţiei 4.2.*) □

• **Schema de principiu a unui automat sincron** va fi, în consecinţă, cea din figura 5.7, în varianta cea mai simplă. Un astfel de circuit elementar va conţine un *CLC* şi un *REGISTRU* ca element de memorare.

Drept consecinţă a segregării funcţionale şi a introducerii semnalului de ceas pentru sincronizare, am obţinut o structură controlabilă atît ca funcţionare cît şi ca proiectare, realizare, testare şi întreţinere. Automatul sincron este varianta inginerească cea mai bine adaptată *contextului tehnologic actual*.

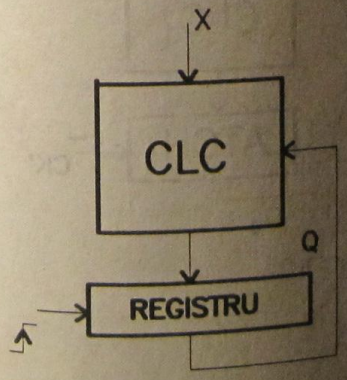



Figura 5.7

### 5.1.3 Structurarea neechilibrată a automatelor sincrone

Configuraţia structurală la care am ajuns mai poate fi îmbunătăţită, chiar dacă realizările concrete actuale nu impun încă acest lucru decît în unele cazuri particulare.

Realizarea unui automat elementar sub forma unui *CLC* şi a unui registru, conectate în buclă, este rezultatul unei opţiuni simplificatoare ce poate fi pusă în evidenţă printr-o detaliere generalizatoare realizată în reprezentarea din figura 5.8. $CLC_0$ şi $LATCH_0$ formează un *SO1* iar $CLC_1$ şi $LATCH_1$ de asemenea. Conectate în buclă generează structura generală a unui *SO2* elementar. *Simetria* structurii ne duce cu gîndul la reprezentările din figura 4.8 care prezentau o simetrie la fel de avansată.

Trecerea de la *SO0* la *SO1* s-a făcut printr-un proces de *structurare echilibrată*, după cum o dovedesc majoritatea circuitelor de memorare anterior studiate. În cazul automatelor finite elementare procesul de structurare nu mai este la fel de echilibrat. Într-adevăr, structura din figura 5.7 se obţine din cea din figura 5.8 suprimînd $CLC_1$. $LATCH_0$ şi $LATCH_1$ rămase se cuplează direct formînd o structură master-slave de *REGISTRU*. Singurul circuit combinaţional este $CLC_0$, care asigură prelucrarea pe buclă. $LATCH_1$ asigură funcţia de memorare propriu-zisă, iar $LATCH_0$ permite netransparenţa pe buclă pentru a se obţine evoluţia.

Acest proces de structurare neechilibrată se va accentua la tranziţiile către ordine superioare lui doi. Menţionăm, de asemenea, că în configuraţii moderne de circuite *VLSI* a început să se manifeste un proces de reechilibrare structurală la nivelul automatelor finite.

Structurarea neechilibrată este utilă, în această etapă, deoarece permite implementări concrete tehnologic eficiente. Limitele introduse de structurările neechilibrate nu sînt majore la nivelul *SO2*.

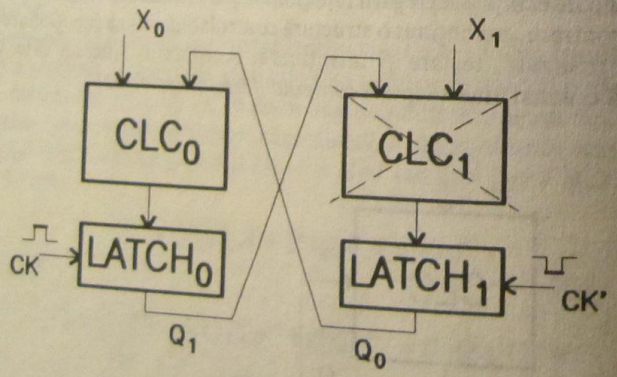



**Figura 5.8**



## 5.2 Automate bistabile

Vom începe studiul automatelor elementare finite cu exemplele cele mai simple, amînînd introducerea lor teoretică pentru a ne familiariza mai întîi cu cîteva circuite concrete.

Circuitele basculante bistabile au fost parţial studiate în capitolul anterior ca sisteme de ordinul unu. Nu am epuizat acolo aceste circuite pentru că o parte dintre ele aparţin sistemelor digitale de ordinul doi. Autonomia unui bistabil de tip *RS* există în sensul că nu urmăreşte orice evoluţie a variabilelor aplicate pe intrare. Dar un astfel de circuit nu poate avea un comportament pe ieşire independent de modificarea valorii variabilelor de intrare. Există şi circuite basculante bistabile a căror evoluţie pe ieşire se manifestă şi în cazuri în care variabila sau variabilele de pe intrare nu-şi modifică valoarea. Este cazul bistabilelor de tip *T* sau *JK*, ce se constituie în cele mai simple automate elementare.

### 5.2.1 Bistabilul T

Bistabilul de tip *T* este cel mai simplu automat ce se poate imagina. Dimensiunea minimă a spaţiului stărilor unui automat este doi. Un automat elementar trebuie să posede cel puţin două stări pentru a putea comuta. Vom nota ieşirea bistabilului cu *Q*, cele două stări fiind $Q=0$ şi $Q=1$. Cea mai simplă comandă ce poate fi dată unui circuit presupune codificarea cu un bit; fie acesta *T*. Celor două valori va trebui să le asociem două comportamente distincte ale automatului, după cum urmează:

- dacă $T=0$, atunci circuitul nu-şi modifică starea;
- dacă $T=1$, atunci circuitul, la fiecare impuls de ceas, îşi modifică starea (dacă este în $Q=0$ trece în $Q=1$ şi invers).

Pentru memorarea stării vom folosi un registru de un bit, deci un bistabil de tip *D*, schema de principiu fiind cea din figura 5.9. *CLC*-ul din figură este comandat

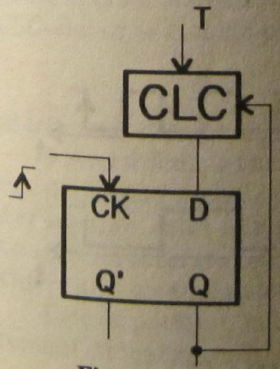



**Figura 5.9**

| T | Q | D |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

**Tabelul 5.1**

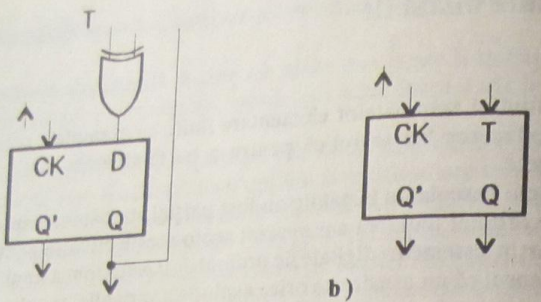



**Figura 5.10**

de intrarea $T$ şi ieşirea $Q$ a bistabilului $D$. Într-adevăr, pentru a-şi modifica starea automatul trebuie să "ştie" în ce stare este; de aici rezultă necesitatea buclei. Tabelul 5.1 defineşte funcţionarea circuitului combinaţional. Pentru $T=0$, $D=Q$, iar pentru $T=1$, $D=Q'$. Rezultă:

$$D = T \oplus Q,$$

ceea ce era de aşteptat, deoarece $Q$ trebuie complementat comandat de variabila $T$ iar *XOR* ştim că poate fi interpretat şi ca un circuit de negare comandat. Schema de principiu a unui bistabil $T$ rezultă ca fiind cea din figura 5.10*a*, iar simbolul logic asociat este cel din figura 5.10*b*.

• Dacă într-o aplicaţie un astfel de circuit trebuie să funcţioneze tot timpul cu $T=1$, atunci $D=Q'$ şi se foloseşte un circuit de tip $D$ cu ieşirea negată $Q'$ conectată la intrare (figura 5.11).

• **Divizarea frecvenţei** unui tren de impulsuri poate fi realizată, cu un circuit bistabil de tip $T$, aplicînd pe intrarea de ceas semnalul de divizat şi culegînd rezultatul la ieşirea $Q$, cu $T=1$. În figura 5.12 este reprezentat modul de funcţionare al circuitului ca divizor de frecvenţă.

Se remarcă faptul că dacă factorul de umplere al semnalului de intrare (*CK*) poate fi oarecare, factorul de umplere al semnalului de ieşire este foarte apropiat

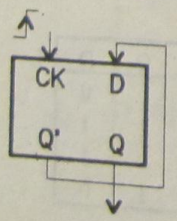



**Figura 5.11**

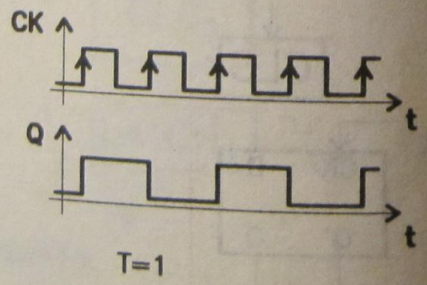



**Figura 5.12**



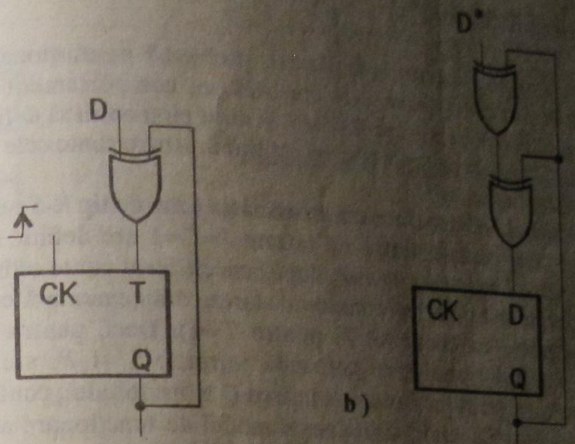



**Figura 5.13**

de *0,5*. Abaterea este dată de diferenţa dintre timpul de propagare la tranziţia pozitivă şi cel asociat tranziţiei negative.

• Acest circuit se mai comportă şi ca un **numărător modulo doi** declanşat de impulsurile de tact. Un numărător modulo doi va număra astfel: zero, unu, zero, unu, zero, ... . Exact la fel "face" ieşirea $Q$ a bistabilului $T$ de cîte ori $T=1$.

• **Transformarea unui bistabil T în bistabil D** este posibilă dacă se observă că:

- pentru a înscrie *0*, bistabilul $T$ trebuie comutat dacă este *1* şi nu trebuie comutat dacă este *0*;
- pentru a înscrie *1*, bistabilul $T$ trebuie comutat dacă este *0* şi nu trebuie comutat dacă este *1*.

Deci avem nevoie de o nouă buclă pentru că va comuta sau nu şi funcţie de starea în care se află. Se deduce imediat că această buclă presupune tot un circuit *XOR*. Rezultă circuitul din figura 5.13*a*. Detaliind obţinem circuitul, din figura 5.13*b*, care este un *SO3*. Nu este nimic anormal, din punct de vedere teoretic, ca funcţia de bistabil de tip $D$ să poată fi regăsită şi în clasa *SO3*, pentru că poate fi regăsită în orice *SON* cu $N \geq 1$. Interesant este, însă, să studiem funcţia de transfer a *CLC*-ului ce se închide peste bistabilul $D$. Din schemă rezultă

$$D = Q \oplus Q \oplus D^* = D^*,$$

deci cele două circuite *XOR* nu fac altceva decît să-şi anuleze reciproc efectul.

• Avertizăm, pe baza acestui exemplu rudimentar, asupra exceselor ce pot fi făcute declanşînd procese de structurare prea complexe. Realizînd o funcţie pe un palier superior celui minim la care poate fi implementată, riscăm să declanşăm procese de structurare inutile. Ceea ce structurăm pe un anumit palier riscă să fie folosit pentru a anihila efectul unor structuri realizate pe un palier inferior.

Este de preferat, în consecinţă, ca o funcţie să fie implementată pe un palier structural cît mai apropiat de cel minim ce-i este asociat.

### 5.2.2 Bistabilul J-K

În capitolul anterior am lăsat o problemă nesoluţionată. Este vorba de bistabilul *R-S* master-slave care nu avea un comportament previzibil pentru configuraţia binară de intrare *R=S=1*. A sosit momentul să definim un bistabil cu două intrări şi cu un comportament utilizabil pentru toate cele patru configuraţii binare aplicabile intrării.

Bistabilul *J-K* funcţionează asemănător celui de tip *R-S* numai că pentru cea de a patra configuraţie binară de intrare *J=K=1* are definit un comportament univoc. Singurul semnificativ este, după cum cititorul poate estima şi singur, acela ca bistabilul, în acest caz, să comute în starea complementară celei în care se află (să se comporte ca bistabilul *T*, pentru *T=1*). Deci, pentru această tranziţie, circuitul combinaţional care comandă intrările *S* şi *R*, ale bistabilului *R-S* master-slave, va primi pe intrare şi ieşirea *Q* a bistabilului, conform reprezentării din figura 5.14*a*. Tabelul 5.2 defineşte modul de funcţionare al acestui automat elementar cu două stări şi doi biţi de intrare, *J* şi *K*.

• Pentru a deduce **structura CLC-ului**, în figura 5.14*b* este definită diagrama *V-K* asociată, ce a fost construită pornind de la definiţia dată de tabelul 5.2. Astfel:

- dacă *J=K=Q=0*, atunci bistabilul poate comuta în *Q=0* (*S=0, R=1*) sau poate să nu comute (*S=0, R=0*), deci *SR=0x*;

- dacă *J=K=0* şi *Q=1*, atunci bistabilul poate să comute în *Q=1* (*S=1, R=0*) sau poate să nu comute (*S=0, R=0*), deci *SR=x0*; ş.a.m.d.

Din diagramă se extrag ecuaţiile

$$S = JQ',$$
$$R = KQ,$$

rezultînd schema din figura 5.14*c* şi simbolul logic al bistabilului *J-K* conform reprezentării din figura 5.14*d*.

• **Autonomia bistabilului J-K** se manifestă prin comportamentul său cînd *J=K=1*. În acest caz el comută la fiecare impuls de ceas, în starea complementară.

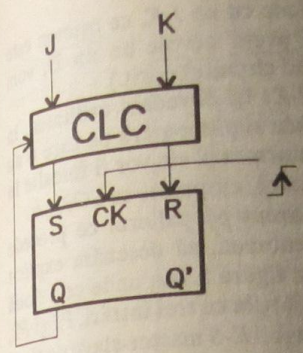
De asemenea autonomia în comportament a permis o simplificare drastică a *CLC*-ului asociat prin apariţia *x*-urilor în diagrama *V-K* asociată. Într-adevăr, dacă automatul "ştie" (datorită buclei) că se află în starea *0*, poate să nu conteze valoarea lui *R* pentru *J=K=1* sau, dacă este în starea *1* şi *J=K=0*, atunci nu contează valoarea lui *S*. Această posibilă lejeritate, vaguitate, în definirea funcţiei de prelucrare pe buclă a condus la simplificarea circuitelor logice de prelucrare. Fără această posibilitate am fi obţinut, spre exemplu, pentru *S* următoarea expresie:

$$S=JK' + JQ'.$$

• **Transformarea în bistabil de tip T** a unui bistabil de tip *J-K* se face, simplu, interconectînd cele două intrări. Deci *T=J=K*. Aceasta este varianta curentă de realizare a unui bistabil *T*, în cataloagele de firmă astfel de circuite nefiind de găsit.

• **Nu este indicată realizarea unui bistabil de tip D** pornind de la unul de tip *J-K* deoarece am realiza o funcţie tipică în clasa *SO1* printr-o structură ce aparţine

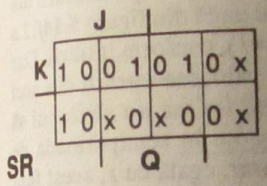




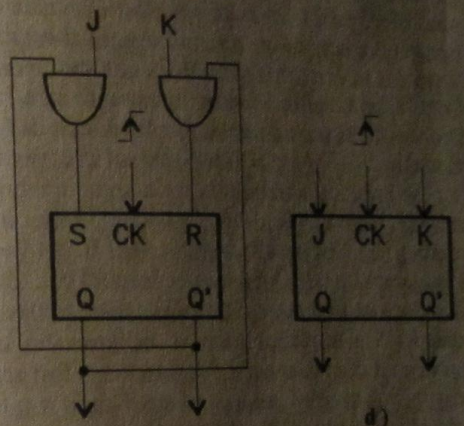

a)

| J | K | Q | |
|---|---|---|---|
| 0 | 0 | Q | nu comută |
| 0 | 1 | 0 | este resetat |
| 1 | 0 | 1 | este setat |
| 1 | 1 | Q' | comută |

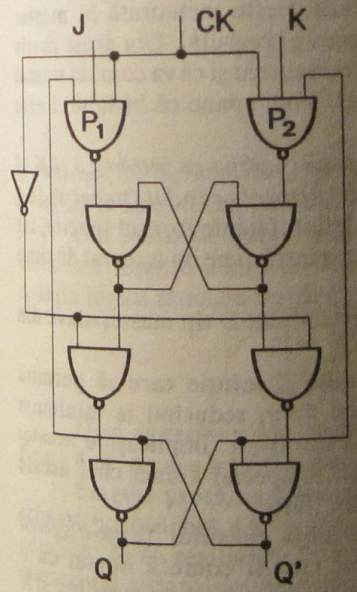
Tabelul 5.2



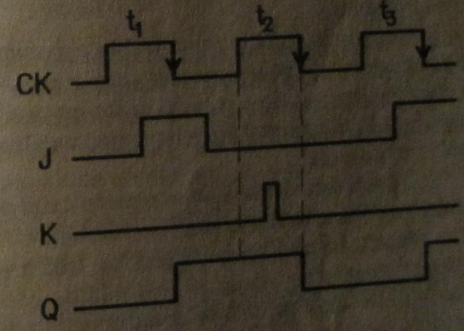

b)




c) d)




e)




f)

Figura 5.14

de *SO2*. (Numai în cazuri limită cînd nu avem ce face cu un *J-K*, ce oricum este implantat pe o plachetă cu circuite integrate, şi avem nevoie de un *D*, vom interconecta între intrările *J* şi *K* un inversor obţinînd circuitul dorit.)

• Bistabilele de tip automat finit (tip *T* şi *J-K*) îşi dovedesc utilitatea, în primul rînd, datorită autonomiei dobîndite prin bucla suplimentară pe care o au faţă de un element de memorare obişnuit. În cele ce urmează ele vor fi folosite în aplicaţii în care-şi vor evidenţia facilităţile: numărătoare, automate ca *SO3*, ş.a.

• **Acest tip de bistabil J-K este semitransparent** pe palierul ce precede frontul activ al ceasului. Pentru a explica acest fenomen, să desenăm explicit structura la care am ajuns. Rezultă schema logică din figura 5.14*e*, unde cele două *AND*-uri explicitate în figura 5.14*c* au fost incluse în porţile cu trei intrări, *P1* şi *P2*, ale latch-ului master. S-a pornit de la schema unui bistabil *R-S* master-slave de tipul celui din figura 4.33, la care au fost eliminate intrările asincrone *R'* şi *S'*. Ieşirea bistabilului comută pe frontul negativ al ceasului, dar această comutare nu este întotdeauna conformă intrărilor *J* şi *K* din "momentul" tranziţiei active a ceasului. Folosim termenul de *moment* în sensul pe care-l acordăm conform reprezentării din figura 4.32. Să considerăm exemplul descris de formele de undă din figura 5.14*f*. La momentul $t_1$, *J=1* şi *K=0* iar bistabilul va comuta în *1* (*Q=1*), conform intrării. Dar în $t_2$ se produce o tranziţie care nu este determinată de valoarea intrării din acel moment. Într-adevăr, la $t_2$, *J=K=0* şi, conform definiţiei, ieşirea nu ar trebui să comute, dar bistabilul comută totuşi în *0* (*Q=0*), deoarece în semiperioada de transparenţă a latch-ului master intrarea *K* a fost, temporar, egală cu *1*, acest fapt înscriindu-se în master şi transferîndu-se în slave la comutarea ceasului. Activarea temporară a intrărilor *J* sau *K* în intervalul în care *CK=1*, acţionează asupra masterului ca şi cum aceste semnale ar fi fost active pe toată durata acestei faze a ceasului. Activarea pentru un scurt timp a intrărilor este memorată în master oricînd pe durata palierului dinaintea frontului activ al ceasului. Din acest motiv nu mai putem spune că bistabilul este complet netransparent şi că va comuta numai conform valorilor intrării din momentul frontului. Vom spune că bistabilul este *semitransparent*.

*Pentru o funcţionare previzibilă va trebui impusă condiţia ca intrările J şi K să fie stabile pe toată durata palierului anterior frontului activ al ceasului* (palierul de *1* în exemplul nostru). Deci, timpul de set-up va fi definit faţă de frontul inactiv, iar cel de hold faţă de frontul activ, impunînd stabilitatea intrării pe un interval de timp supărător de mare.

O problemă similară nu se pune în cazul bistabilului *D* tip master-slave din motive ce credem că sînt evidente cititorului.

Pentru bistabilele de tip *J-K* va trebui găsită o soluţie care să permită *definirea timpilor de set-up şi hold faţă de acelaşi front*, reducînd la minimum intervalul de timp în care se impune stabilitatea intrărilor. Implicaţiile acestei restricţii suplimentare vor fi evidente în aplicaţii, în special atunci cînd aceste circuite vor fi folosite pentru realizarea automatelor finite.

• **Bistabilul J-K activat exclusiv de tranziţia ceasului** (*edge-triggered flip-flop*) este reprezentat în figura 5.15*a*. Acest circuit comută numai ca o consecinţă a tranziţiei negative a ceasului şi a valorilor pe care le au intrările *J* şi *K*



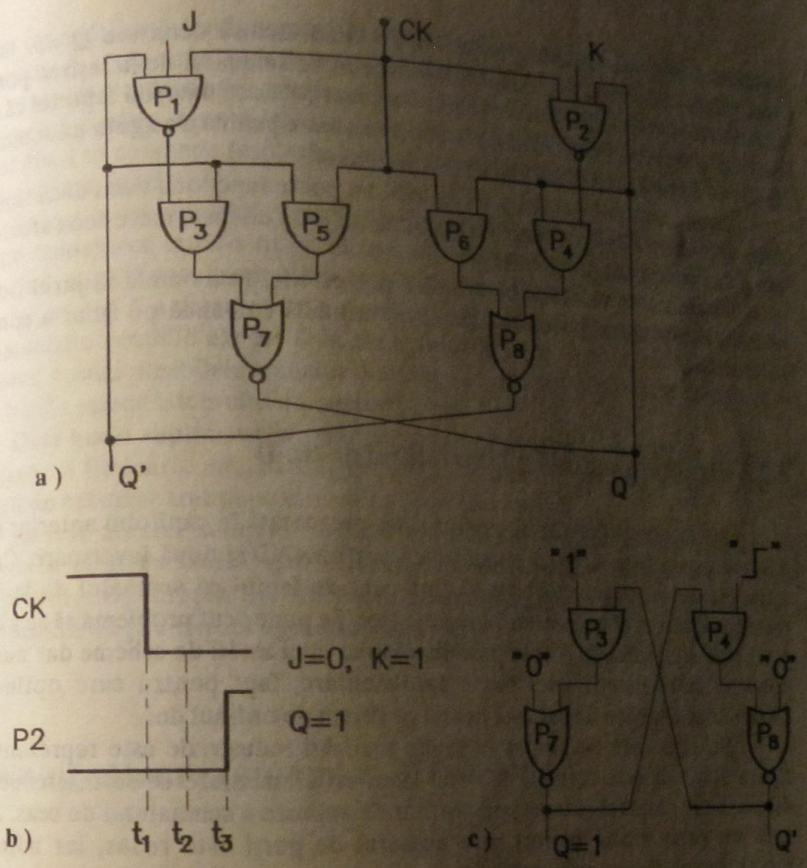



**Figura 5.15**

în "momentul" acestei tranziţii. *Stabilitatea intrărilor J şi K este impusă printr-un timp de set-up şi unul de hold asociate frontului negativ al ceasului.* Să explicăm!

Din punct de vedere static nici una din cele trei intrări ale bistabilului nu afectează ieşirile acestuia. Afirmaţia anterioară poate fi uşor probată scriind ecuaţia logică asociată ieşirii *Q*, după cum urmează:

$$Q = (CK \cdot Q' + Q'(CK \cdot J \cdot Q'))' =$$
$$= (Q'(CK + (CK)' + J' + Q))' = (Q')'.$$

Deci, *în regim static* schema de principiu a bistabilului este de tipul celei din figura 4.6*c*, nu are intrări şi nu poate fi acţionată în nici un fel.

Singura şansă, ca schema să poată fi comutată, o avem *în regim dinamic* cînd schema echivalentă s-ar putea să fie alta, datorită propagărilor, prin porţile ce o compun, ce se realizează în intervale de timp diferite de zero. Într-adevăr, ca o consecinţă a tranziţiei negative a ceasului, pentru *J=0*, *K=1* şi *Q=1*, în intervalul de timp dintre $t_2$ şi $t_3$, conform figurii 5.15*b*, schema echivalentă a bistabilului este cea reprezentată în figura 5.15*c*, dacă timpul de propagare prin *P2* este $t_3$-$t_1$. În

intervalul $t_2$-$t_1$ se închide poarta *P6*, *P5* fiind închisă deoarece $Q'=0$. Deci, în intervalul de timp $t_3$-$t_2$ bistabilul este resetat de semnalul de la ieşirea porţii *P2*, basculînd în starea $Q=0$. Comutarea s-a putut produce datorită faptului că *timpul de propagare prin poarta P2 a fost suficient de mare* pentru a asigura un semnal cu o astfel de durată încît bistabilul să poată comuta.

Deci, atenţie! Acest tip de bistabil nu poate funcţiona decît dacă timpul de propagare prin *P2* şi *P1* (pentru comutarea în $Q=1$) este mai mare decît al celorlalte porţi din componenţa schemei.

Stabilitatea valorilor binare pe *J* şi *K* este impusă numai în jurul frontului activ al ceasului, asigurîndu-se astfel o veritabilă comandă pe front a comutării bistabilului.

### 5.2.3 Structura optimă a bistabilului de tip D

Structura de bistabil *D* master-slave prezentată în capitolul anterior (figura 4.34) se poate realiza cu un număr de 8 porţi *NAND* şi două inversoare. Calculul timpului de propagare trebuie să ţină cont de faptul că semnalul de la intrare trebuie să parcurgă minimum 5 nivele logice. Se pune acut problema să se găsească o schemă simplificată. A fost posibilă găsirea unei astfel de scheme dar numai cu condiţia introducerii unei bucle suplimentare, fapt pentru care optimizarea bistabilului *D* poate fi realizată numai ca sistem de ordinul doi.

Schema care s-a impus în multe realizări industriale este reprezentată în figura 5.16; ea este formată din trei latch-uri elementare ce se interconectează direct, între ele, fără intervenţia porţilor de aplicare a semnalului de ceas. Acesta este un prim motiv pentru care numărul de porţi este redus, iar adîncimea circuitului va fi şi ea micşorată.

Bistabilul comută drept consecinţă a tranziţiei pozitive a ceasului. Dacă $CK=0$, atunci ieşirile *A* şi *B* ale primelor două latch-uri sînt forţate pe *1*, indiferent de valoarea aplicată pe intrarea *D*. Latch-ul de ieşire este, astfel, neacţionat, conservîndu-şi starea în care a fost lăsat de o comandă anterioară. În "momentul" comutării ceasului din *0* în *1* intrarea *D* poate avea *0* sau *1*. Dacă $D=1$, ieşirea *A* rămîne nemodificată iar *B* comută în *0*, comandat de starea latch-ului anterior. Comutarea lui *B* în zero setează latch-ul de ieşire ($Q=1$). Deci *Q* ia valoarea lui *D* drept consecinţă a frontului pozitiv al ceasului *CK*. Dacă $D=0$, în acelaşi moment al frontului pozitiv, atunci ieşirea *A* va comuta în *0*, *B* rămînînd neschimbată. Zeroul din *A* va comanda resetarea latch-ului de ieşire ($Q=0$), comutînd bistabilul, astfel, conform intrării *D*.

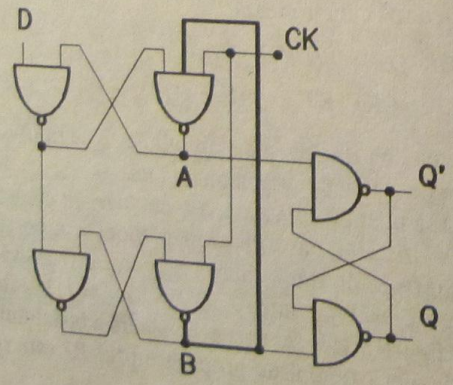



**Figura 5.16**

Care este rolul buclei ce porneşte de la ieşirea *B*? Această conexiune nu va permite comutarea latch-ului de intrare sub influenţa lui *D*, cînd $CK=1$. Riscul apare cînd, după frontul activ, *D* ar comuta în zero. Dacă *D* a fost *1* în "momentul" frontului, atunci *B* a comutat în *0*. Acest zero fixează pe *A* la valoarea *1* indiferent de comutările posibile ale lui *D* în zero. Dacă *D* a fost zero la tranziţia activă a frontului, atunci posibilele tranziţii ulterioare în unu sînt inofensive pentru că nu pot schimba starea latch-ului de intrare ($A=0$).

Deci bucla suplimentară, marcată îngroşat în figură, permite ca timpul de hold ($t_H$) să fie foarte mic. Se asigură, astfel, comutarea pe front a bistabilului în condiţii de netransparenţă pe unul din palierele ceasului.

Rostul acestei scheme, faţă de cea anterior studiată, este asigurat de simplitatea ei şi de viteza sporită de comutare. Ne aflăm în faţa unui rarisim exemplu în care creşterea ordinului sistemului asigură optimizarea atît a complexităţii cît şi a vitezei unei structuri digitale.

## 5.3 Numărătoare (automate $2^n$ - stabile)

Una din funcţiile cele mai importante ce se regăseşte în sistemele digitale este asociată funcţiei aritmetice celei mai simple: *incrementarea*, adică adunarea cu o unitate. Incrementarea se obţine cel mai simplu prin *numărare*. Structurile fizice ce realizează această funcţie se numesc *numărătoare*. Ele sînt automate finite deoarece primesc comanda simplă: "adună la numărul curent o unitate" şi generează *pornind de la starea* (numărul) *curentă*, starea următoare ce reprezintă numărul incrementat.

Schema simplificată a unui numărător este dată în figura 5.17, unde în fiecare ciclu de ceas în care $U=1$ (*U* de la *up*), valoarea numerică reprezentată de

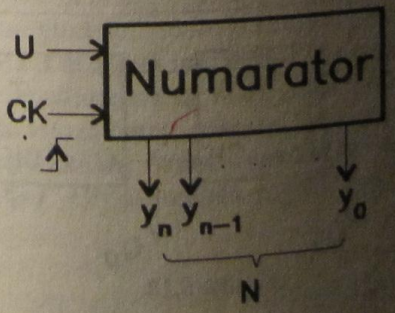



**Figura 5.17**

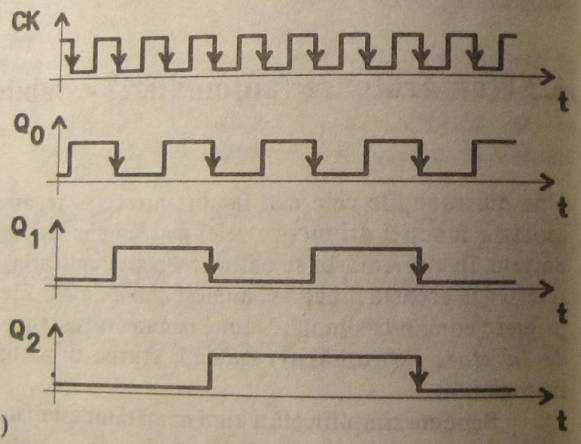
configuraţia binară $y_n y_{n-1} \dots y_0$ este incrementată. Această configuraţie binară reprezintă numărul $N$. Numărul maxim ce poate fi reprezentat în acest automat (numărător) este $2^{n+1}-1$. Numărarea se face *modulo* $2^{n+1}$. Odată atins numărul maxim ce poate fi reprezentat, la o nouă comandă de numărare ($U=1$) numărul $N$ ia valoarea zero.

### 5.3.1 Numărătoare asincrone

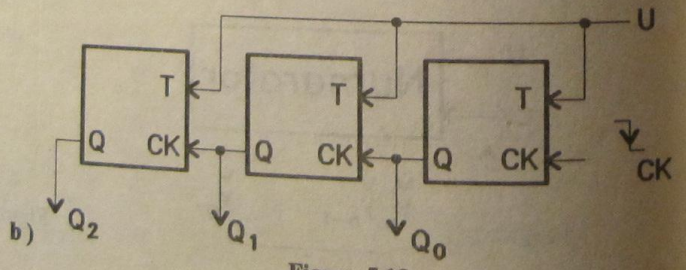
Vom încerca să arătăm, pentru început, că divizarea multiplă a frecvenţei este echivalentă cu numărarea binară. Am văzut, în cazul bistabilului $T$, că *divizarea frecvenţei* este realizată cu acelaşi circuit care realizează şi *numărarea modulo doi*. Tabelul 5.3 reprezintă o secvenţă de numărare modulo *8* realizată cu configuraţia binară $Q_2Q_1Q_0$. În paralel, în figura 5.18*a* sînt reprezentate semnalele obţinute prin divizarea de trei ori a unui semnal periodic, *CK*. Iniţial $Q_2=Q_1=Q_0=0$; la aplicarea primului front negativ rezultă configuraţia binară $Q_2Q_1Q_0=001$, iar în continuare *010* ş.a.m.d. pînă la *111* după care, la următorul impuls de ceas (tact) se revine la configuraţia *000* reluîndu-se ciclul. Remarcăm coincidenţa dintre o secvenţă de numărare binară (tabelul 5.3) şi semnificaţia ce poate fi asociată unor trenuri de impulsuri obţinute printr-o succesivă divizare a frecvenţei. Rezultă că *un numărător este un divizor multiplu de frecvenţă* ce se obţine prin cascadarea unor divizoare de frecvenţă (realizabile cu cîte un bistabil de tip $T$).

| $Q_2$ | $Q_1$ | $Q_0$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |
| 1 | 1 | 1 |

Tabelul 5.3




a)




b)

Figura 5.18

• **Numărătorul** va putea fi structurat conform reprezentării din figura 5.18*b*. Pentru $U=1$ funcţionarea acestui circuit este descrisă de formele de undă din figura 5.18*a*. Fiecare bistabil $T$ formează un divizor de frecvenţă ce generează semnale de ceas divizate pentru următorul divizor. Dacă $U=0$ starea bistabilelor nu se schimbă.

După fiecare impuls de ceas numărul format de biţii $Q_2$, $Q_1$ şi $Q_0$ creşte cu o unitate modulo doi, dacă $U=1$.

Creşterea capacităţii de numărare a numărătorului se poate face după o regulă foarte simplă: se mai adaugă un bistabil de tip $T$.

• **Numărarea inversă** se poate obţine de la aceeaşi structură considerînd, ca ieşiri ale numărătorului, ieşirile de tip $Q'$ ale fiecărui bistabil. Acest lucru se poate explica foarte simplu complementînd bit cu bit configuraţiile binare din tabelul 5.3 sau negînd formele de undă din figura 5.18*a*. Deci un numărător care numără invers se poate obţine cu acelaşi tip de circuit, dar cu unul distinct. Pentru a construi unul care poate număra, comandat, şi înainte şi înapoi, circuitul va trebui complicat suplimentar. Lăsăm cititorului, drept exerciţiu, acest efort.

• **Timpul de numărare** este intervalul de timp scurs de la frontul activ pînă în momentul în care au comutat toate $T$-urile generînd noua configuraţie binară, cea incrementată. Dacă timpul de propagare de la *CK* la $Q$ printr-un bistabil $T$ este $t_{pT}$, atunci acest timp este, în situaţia cea mai defavorabilă

$$T(n) = (n+1)t_{pT}$$

pentru un numărător cu $n+1$ celule. Deci

$$T(n) = O(n).$$

Într-adevăr, cînd numărătorul din figura 5.18*b* comută din *111* în *000*, configuraţia finală se stabileşte prin comutarea tuturor bistabilelor de tip $T$. Primul comută în $t_{pT}$, al doilea primeşte comandă după $t_{pT}$ şi comută în $t_{pT}$ ş.a.m.d. Deci numai după $3t_{pT}$ numărătorul comută din *111* în *000*. La un numărător ce numără înapoi acest caz limită, cel de care vom ţine cont la estimarea vitezei de lucru a circuitului, se produce la tranziţia din *000* în *111*.

• Nu toţi bistabilii numărătorului comută în acelaşi timp, nefiind comandaţi de acelaşi impuls pe intrarea de ceas. Din acest motiv aceste numărătoare se numesc *numărătoare asincrone*. Perioada minimă a impulsurilor de ceas va trebui astfel dimensionată încît şi în cazuri limită să se poată produce, între două fronturi active, stabilizarea configuraţiei binare şi să rămînă un interval de timp suficient pentru ca aceasta să poată fi utilizată în comandarea altor circuite. Deci: $T_{min} > (n+1)t_{pT}$.

Pentru numărătoare de mare capacitate soluţia asincronă oferă circuite cu performanţe de viteză foarte scăzute. Singurul lor avantaj este simplitatea.

• **Complexitatea unui numărător asincron** este în mod evident $S_{Num_A}(n) = O(n)$. De asemenea regula după care creşte structura este foarte simplă.

### 5.3.2 Numărătoare sincrone

Dezavantajul unui timp de numărare $O(n)$ impune căutarea unor soluţii alternative pentru creşterea vitezei acestor circuite. O soluţie drastică o constituie numărătoarele sincrone, astfel concepute încît să permită comanda sincronă a tuturor bistabilelor ce-l compun.

Principiul ce stă la baza numărătoarelor sincrone diferă de cel ce a permis realizarea numărătoarelor asincrone. Să considerăm configuraţiile binare codificate prin $Q_n Q_{n-1} \dots Q_0$. În procesul de incrementare al unei configuraţii binare oarecare *bitul $Q_i$ se complementează dacă şi numai dacă toţi biţii de rang inferior sînt egali cu 1*, deci $Q_j=1$ pentru $j<i$. Inspectînd configuraţiile binare din tabelul 5.3 putem să verificăm şi să justificăm afirmaţia anterioară.

• **Structura de principiu a unui numărător strict sincron** se bazează pe faptul că detectăm, printr-un circuit *AND* cu $i$ intrări, provenind de la ieşirile bistabililor de rang inferior, condiţia ca bistabilul de rang $i$ să comute. Figura 5.19 reprezintă un numărător sincron cu $n+1$ celule de numărare realizat prin interconectarea în cascadă a $n+1$ bistabile de tip $T$ comandate conform regulii anterior enunţate. Acest circuit reprezintă o *extensie serie* în clasa *SO2*. A fost adăugată cîte o intrare la fiecare poartă pentru a controla numărarea prin semnalul $U$ (*up*) aplicat pe aceste intrări suplimentare. Dacă $U=0$ atunci starea bistabilelor nu se poate schimba, iar pentru $U=1$ comută acele bistabile ce sînt precedate de bistabile ce se află toate în starea *1*. Rezultă că *bistabilele comută sincron cu frontul activ al ceasului CK*.

Viteza maximă cu care poate să funcţioneze un astfel de numător este dată de timpul de propagare printr-un bistabil $T$, de timpul de propagare printr-o poartă şi de timpul de set-up pe intrarea bistabililor. Teoretic, afirmaţia anterioară este valabilă riguros. Practic, pentru $n$ foarte mare, capacitatea parazită echivalentă ce încarcă ieşirile bistabilelor de ordin inferior este foarte mare, datorită faptului că aceste ieşiri sînt conectate la un număr foarte mare de intrări. Deci timpul de comutare a semnalului la ieşirea acestor bistabili devine semnificativ, adăugîndu-se celui de propagare prin bistabil. Rezultă o scădere suplimentară a vitezei maxime de funcţionare a numărătorului, pentru $n$ ce depăşeşte o anumită limită, dată de tehnologia de realizare a circuitelor elementare.

Presupunerea că fan-out-ul şi fan-in-ul circuitelor poate fi oricît de mare nu este practic valabilă, rezultînd necesitatea unor structuri sau structurări suplimentare care măresc complexitatea şi adîncimea circuitului.

• **O variantă de numărător sincron**, ce presupune circuite cu fan-in şi fan-out finite, este reprezentată în figura 5.20. Funcţionarea acestui circuit este evidentă dacă a fost înţeleasă cea a numărătorului anterior prezentat. În acest caz s-a pornit de la echivalenţa evidentă

$$Q_0 \cdot Q_1 \cdot \dots \cdot Q_i = (\dots((Q_0 \cdot Q_1) \cdot Q_2) \dots) Q_i$$

care presupune realizarea unui *AND* cu $n$ intrări cu ajutorul a $n-1$ *AND*-uri cu *fan-in* = 2. Acest mod de abordare prezintă şi avantajul unei complexităţi reduse deoarece circuitele ce contribuie la realizarea circuitului cu funcţia $Q_0 \cdot Q_1 \cdot \dots \cdot Q_i$ vor fi folosite integral şi pentru circuitul asociat expresiei $Q_0 \cdot Q_1 \cdot \dots \cdot Q_{i+1} \cdot$



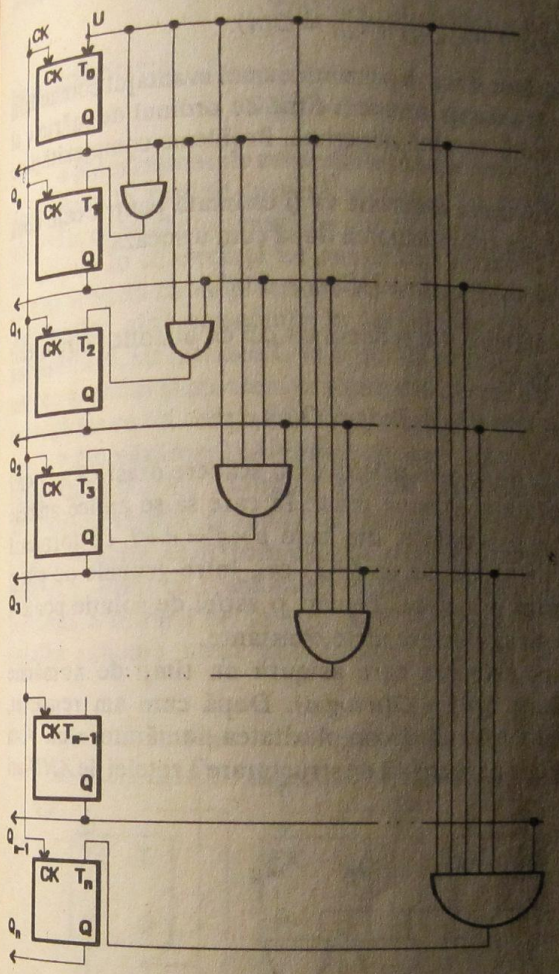



**Figura 5.19**

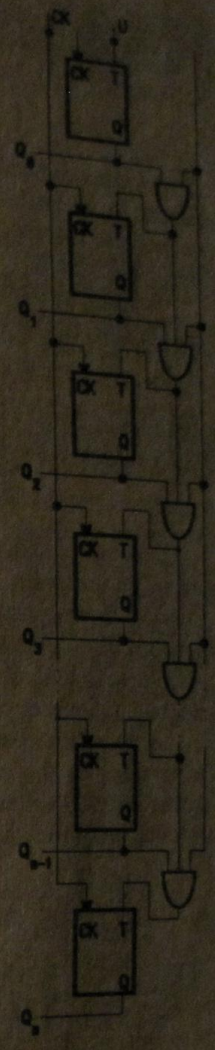



Figura 5.20

Dezavantajul major al acestei variante este dat de limitarea drastică a frecvenţei de lucru pentru cazul în care $n$ devine prea mare.

Se pot imagina şi variante intermediare în care pe subşiruri de bistabile $T$ să se aplice primul principiu, iar la înlănţuirea lor, pentru a forma un numărător mai mare, să se folosească cel de al doilea procedeu de interconectare.

• **Timpul necesar unei incrementări** impune valoarea minimă a perioadei ceasului. Pentru numărătorul din figura 5.19

$$T_1(n) = t_{pT} + t_{pAND} + t_{SU} = O(1),$$

fiind dat, deci, de timpul de propagare prin bistabilul $T$ şi prin poarta *AND*, la care se adaugă timpul de set-up pe intrarea bistabilului. Pentru numărătorul din figura 5.20 scriem

$$T_2(n) = t_{pT} + n\,t_{pAND} + t_{SU} = O(n).$$

A doua variantă de numărător sincron prezintă numai avantajul comandării sincrone a tuturor bistabilelor, frecvenţa de lucru fiind de ordinul de mărime al frecvenţei de lucru asociate numărătoarelor asincrone. Problema complexităţii se pune, iarăşi, diferit.

• **Complexitatea numărătoarelor sincrone** va fi estimată pentru cele două variante separat. Prima variantă are complexitatea după cum urmează

$$S_1(n) = c_1(n+1) + c_2((n+3)/2)n = O(n^2),$$

primul termen fiind asociat bistabilelor din schemă iar cel de al doilea circuitelor *AND*. Pentru a doua variantă scriem

$$S_2(n) = c_1(n+1) + 3nc_2 = O(n).$$

Reducerea complexităţii la $O(n)$ este plătită cu o scădere drastică a vitezei maxime de numărare. Se pot concepe scheme mixte în care să se aplice ambele procedee. Pentru grupe de cîte $p$ bistabile, din cele $p \times q = n+1$ ce formează numărătorul, se aplică primul principiu iar de $(q-1)$ ori, între grupele de cîte $p$ bistabile, se aplică cel de al doilea principiu. Tehnic, o astfel de soluţie poate fi acceptată, teoretic ea nu face decît să afecteze nişte constante.

• **O soluţie intermediară** este cea care asigură un timp de numărare $T(n) = O(\log n)$ şi o complexitate $S(n) = O(n \log n)$. După cum am remarcat, reţeaua de *AND*-uri este cea care determină complexitatea numărătorului. Vom prezenta, în figura 5.21 o modalitate particulară de structurare a reţelei de *AND*-uri

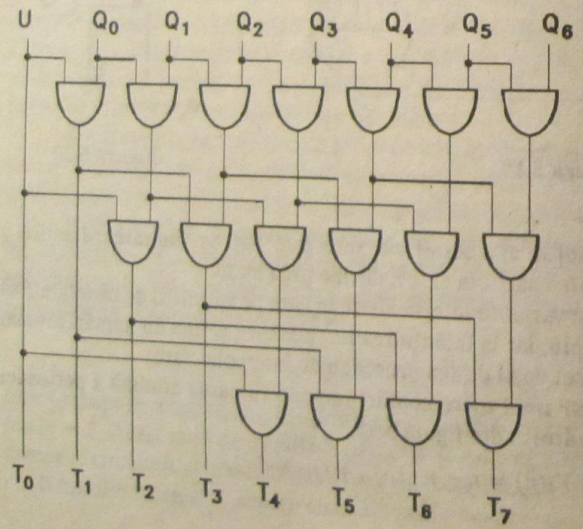



**Figura 5.21**



care asigură un timp şi o complexitate ce corespund celor anterior enunţate. Reprezentarea a fost făcută pentru $(n+1) = 8$. Adîncimea reţelei de acest tip este, evident

$$T(n) = t_{pAND} \log_2 n = O(\log n)$$

iar complexitatea

$$S(n) = 1{\cdot}n + n \log_2 n = O(n \log n).$$

• **Numărătoarele reversibile** funcţionează pe un principiu asemănător. Dacă parcurgem în ordine inversă secvenţa de configuraţii binare din tabelul 5.3, observăm că bitul $Q_i$ comută de cîte ori toţi cei anteriori sînt pe zero, iar $Q_0$ comută tot timpul. În consecinţă un numărător reversibil (care numără înapoi) are exact aceeaşi schemă ca unul ce numără înainte (figura 5.19) cu deosebirea că intrările *AND*-urilor sînt conectate la ieşirile de tip $Q_i'$ ale bistabililor ce compun numărătorul, iar intrarea $U$ va fi notată prin $D$ (*down*). Sau, putem lăsa conexiunile interne aceleaşi şi scoatem la ieşire din fiecare bistabil borna $Q'$. Într-adevăr, dacă se neagă bit cu bit conţinutul tabelului 5.3 se obţine o secvenţă de numărare înapoi.

• **Numărătorul up-down** se poate construi pornind de la observaţia facută în paragraful anterior adăugînd $n$ *MUX*-uri de două căi care selectează la intrările *AND*-urilor ieşirile $Q$ sau $Q'$ ale bistabilelor de tip $T$. (Aceste *MUX*-uri realizează de fapt, funcţia *XOR* prin care se complementează comandat ieşirea $Q$ a fiecărui bistabil.) Bitul ce comandă numărarea înainte sau înapoi va fi folosit ca bit de comandă a intrării de selecţie a *MUX*-urilor. În figura 5.22 este reprezentată în detaliu schema unui numărător up/down de patru biţi. Intrarea *COUNT*

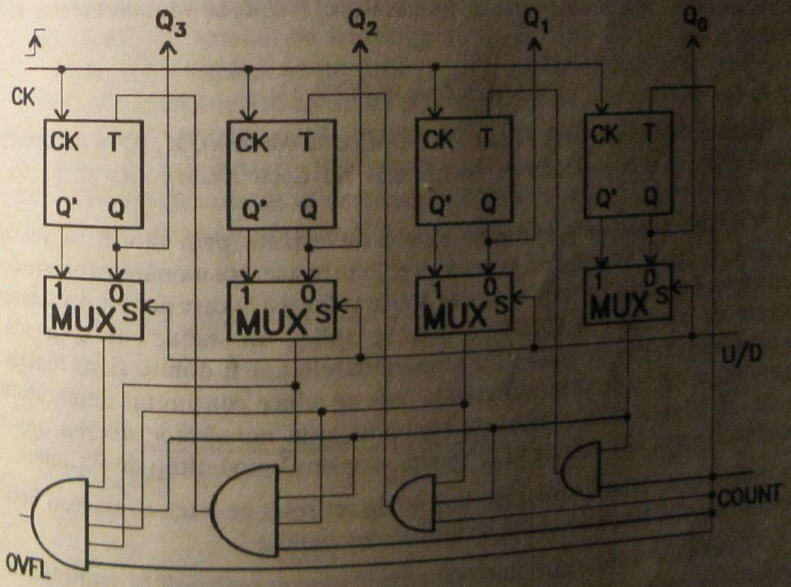



**Figura 5.22**

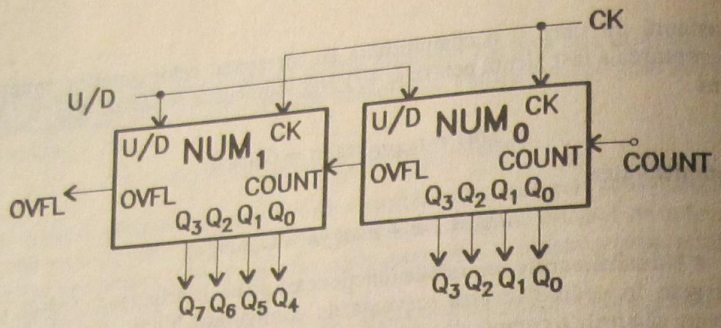



Figura 5.23

condiţionează numărarea iar intrarea *U/D* condiţionează sensul în care se numără. Ieşirea *OVFL* (*overflow*) indică *depăşirea capacităţii de numărare* a circuitului, fiind folosită pentru structurarea unui circuit de numărare cu capacitate mai mare. Astfel, în figura 5.23 este prezentat modul în care poate fi conceput un numărător de opt biţi (numărător modulo $2^8$) prin interconectarea a două numărătoare de patru biţi de tipul celor din figura 5.22. Intrările *CK* şi *U/D* de la cele două numărătoare se vor conecta împreună. Ieşirea *OVFL* de la $NUM_0$ se aplică la *COUNT* de la $NUM_1$ determinînd o incrementare sau o decrementare a $NUM_1$ numai cînd este depăşită capacitatea de numărare a $NUM_0$. Spre exemplu, de cîte ori $NUM_0$ este în starea

$$Q_3Q_2Q_1Q_0 = 1111$$

şi este comandată numărarea înainte, numărătorul $NUM_1$ se va incrementa, $NUM_0$ trecînd în starea

$$Q_3Q_2Q_1Q_0 = 0000.$$

În cazul decrementării, de cîte ori $NUM_0$ este *0000*, $NUM_1$ se va decrementa. Evident, $NUM_1$ va fi acţionat de *16* ori mai rar decît $NUM_0$ într-un proces de numărare continuă.

• **Numărătoarele presetabile** sînt caracterizate prin faptul că valoarea înscrisă în ele poate fi încărcată din exterior. Pentru aceasta numărătorul este dotat cu intrări suplimentare pe care se poate aplica valoarea la care are loc presetarea şi o intrare de comandă (*LOAD*) pe care se aplică un semnal de comandă ce declanşează operaţia. De multe ori numărătoarele pot fi dotate şi cu funcţia de ştergere (*CLEAR*) prin care valoarea la care se aduce conţinutul numărătorului este *00...0*. Figura 5.24 reprezintă simbolul unui *numărător sincron up/down presetabil* de patru biţi. Intrarea *LD'* poate avea două modalităţi de acţiune:

- determină o încărcare asincronă de tip set/reset pe latch-ul slave al fiecărui bistabil, acţionînd pe palierul impulsului de comandă;

- determină o încărcare sincronă ce este efectiv realizată pe frontul activ al semnalului de ceas (*CK*).



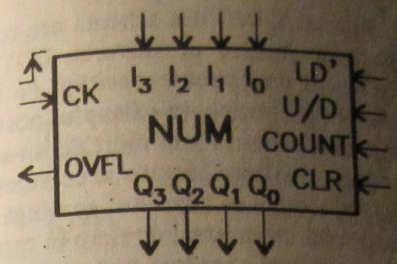



Figura 5.24

În primul caz bistabilii se comportă ca *latch*-uri transparente pe durata palierului activ al comenzii *LD'*, iar în al doilea caz bistabilii numărătorului se comportă ca un *registru* realizat cu structuri master-slave. În primul caz peste numărător nu va putea fi închisă o buclă controlată, pe cînd în cel de al doilea caz acest lucru va fi posibil.

## 5.3.3 Divizoare programabile

Ieşirea $Q_i$ a unui numărător generează un semnal cu o frecvenţă de $2^{-(i+1)}f_{CK}$, unde $f_{CK}$ este frecvenţa impulsurilor de ceas. Putem obţine astfel divizări cu puteri ale lui doi. Se pune problema să generăm semnale cu o frecvenţă $f_{CK}/N$, unde $N$ este un *întreg oarecare*. Altfel spus, să construim, cu ajutorul numărătoarelor presetabile, *divizoare programabile*.

• **Schema de principiu a unui divizor programabil** este dată în figura 5.25. Pe intrarea *CK* se aplică un semnal periodic cu frecvenţa de repetiţie $f_{CK}$ şi se obţine pe ieşirea *OVFL* un semnal cu frecvenţa de $f_{CK}/N$ unde $N$ este un întreg strict pozitiv. Numărul $N$ se aplică pe intrările $I_n$ ... $I_0$, codificat binar. Numărătorul este astfel comandat, încît numără continuu şi înapoi (*U/D* = *COUNT* = *1*). Odată ajuns la valoarea zero, se activează ieşirea *OVFL* şi este comandată încărcarea numărătorului cu numărul $N$. Se dezactivează astfel ieşirea *OVFL* şi dispărînd comanda *LD'* numărătorul se va decrementa pînă va ajunge iarăşi la valoarea zero. Astfel *OVFL* se activează odată la $N$ impulsuri de ceas, apariţia lui producîndu-se

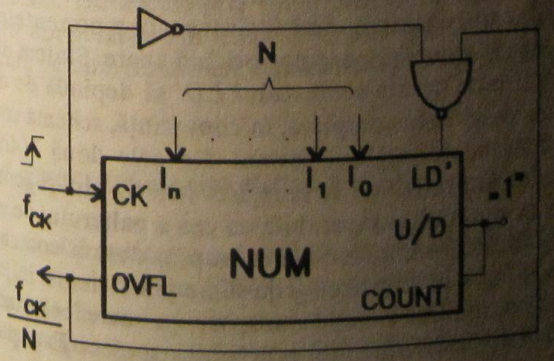



Figura 5.25

cu frecvenţa anterior specificată. Poarta *NAND* din schemă are rolul de a evita aplicarea semnalelor parazite date de hazard pe intrarea *LD'*. *NAND*-ul ce generează semnalul *OVFL* semnalizează depăşirea capacităţii de numărare, dar imediat după frontul activ al ceasului (cel pozitiv în exemplul nostru) pot apare semnale datorate hazardului atunci cînd comută mai mult de un singur bistabil. Poarta adiţională se va deschide numai pe palierul negativ al ceasului la un interval de timp suficient de mare astfel încît procesele de hazard să se stingă. Frontul activ fiind cel pozitiv, imediat după el urmează un interval de timp în care acest *NAND* adiţional este închis. În funcţionarea concretă apar însă nişte dificultăţi.

• **Implementarea divizoarelor programabile cu numărătoare presetabile asincron** se cere analizată cu atenţie pentru că practic apar dificultăţi ce implică o completare a schemei de principiu.

Pornim de la observaţia că semnalul *OVFL* este activ atît timp cît valoarea încărcată în bistabilele numărătorului este zero. În acelaşi timp acest semnal acţionează astfel încît valoarea încărcată în numărător devine diferită de zero, creînd condiţiile ca el să dispară. *Apariţia* semnalului *OVFL* determină imediat, prin procese de propagare necontrolată, *dispariţia* sa. Deci *durata* acestui semnal nu este determinată de perioada ceasului ci de propagarea prin structuri combinaţionale. Se închide, de fapt, o buclă de tip asincron de tipul celei evidenţiate în subcapitolul 5.1.

Efectul cel mai neplăcut nu este datorat exclusiv faptului că durata acestui impuls este mică şi necontrolabilă. Necazul cel mai mare este că efectele acestui impuls asupra numărătorului nu sînt în totalitate previzibile. Într-adevăr, pentru ca *OVFL* să dispară este suficient ca un singur bistabil să se încarce. Dispariţia acestui semnal nu înseamnă că toţi bistabilii ce trebuiau încărcaţi cu *1* au comutat, ci numai faptul că cel puţin unul a comutat. Într-un numărător se pot găsi oricînd bistabile ce comută mai lent, bistabile pentru care durata impulsului de încărcare trebuie să fie mai lungă. Un astfel de bistabil va fi "dezavantajat" de compania altuia, mai vioi, care comutînd rapid scurtează drastic durata semnalului *OVFL* sub limita la care acesta poate fi perceput de bistabilii mai leneşi. Un bistabil poate comuta mai lent şi datorită încărcării (fan-out-ului) mai mari pe care o are în schema internă sau externă numărătorului. Configuraţia internă a circuitului pune clar în evidenţă încărcarea progresiv crescută a structurilor asociate ordinelor binare inferioare.

În consecinţă, ne putem aştepta ca schema din figura 5.25 să funcţioneze uneori eronat, iar dacă un exemplar experimental nu dă erori acest fapt nu este semnificativ deoarece prin multiplicare, îmbătrînirea componentelor sau schimbarea temperaturii, erorile în funcţionare pot oricînd apare. Sîntem obligaţi să concepem o schemă astfel încît durata semnalului *LD'* să depindă de durata, controlabilă, a impulsului de ceas. Vom completa, în consecinţă, schema conform reprezentării din figura 5.26, unde latch-ul format din cele două *NAND*-uri "zăvorăşte" semnalul cu durată imprevizibilă de la ieşirea *NAND*-ului şi generează pe intrarea *LD'* un semnal cu o durată comparabilă cu cea a palierului de zero al ceasului. Chiar dacă unii bistabili vor comuta mai rapid, durata comenzii de încărcare va fi fixă, iar prin proiectare vom avea grijă să fie şi suficientă.

Latch-ul suplimentar are rolul de a anula transparenţa buclei ce se închidea peste schema din figura 5.25. Bucla devine astfel previzibilă ca efect. Este şi acesta



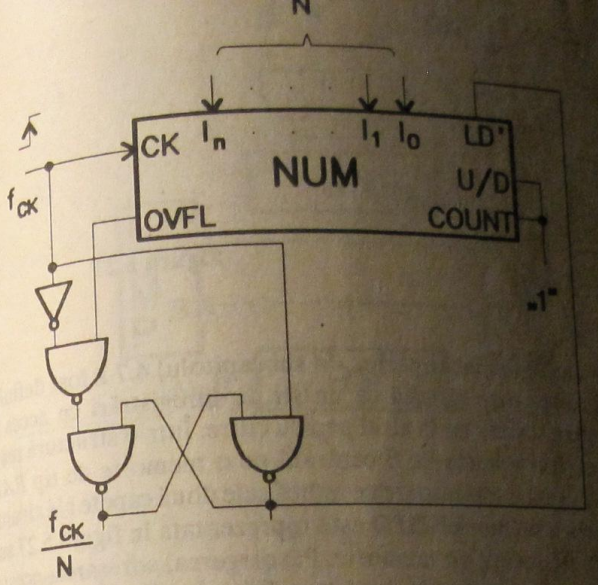



Figura 5.26

un bun exemplu pentru a ilustra efectul buclelor combinaţional transparente. Rezolvarea este iarăşi tipică deoarece latch-urile slave din numărător constituie "un master" pentru latch-ul adiţional, formînd astfel o structură master-slave.

• **Realizarea divizoarelor programabile cu numărătoare sincron presetabile** este posibilă conform unei scheme simplificate în care ieşirea *OVFL* se conectează *direct* la intrarea *LD*. Această din urmă intrare odată activată, încărcarea se realizează cu următorul front activ aplicat intrării de ceas. Durata semnalului este dată în acest caz de perioada ceasului.

• **Hazardul de la ieşirea divizorelor programabile**, în cazul schemei din figura 5.25 sau altora similare, se datorează modului pur combinaţional de formare a semnalului *OVFL*. Din acest motiv, în schemele concrete, acest semnal este uneori condiţionat de impulsul de tact (a se vedea în acest sens, în catalog, schema concretă a numărătorului 74193).

• În funcţie de schema concretă utilizată, atenţionăm asupra faptului că mărimea *N* aplicată la intrare poate să difere cu o unitate pentru buna funcţionare a schemei. De asemenea, funcţionarea la frecvenţe limită trebuie analizată cu atenţie din perspectiva timpilor de set-up ce caracterizează relaţia între semnalele aplicate pe diferite intrări de comandă ale numărătorului.

## 5.3.4 Structură de memorie FIFO

O altă aplicaţie a numărătoarelor este implementarea concretă eficientă a funcţiei de memorie *FIFO*. Am arătat că această funcţie este definită şi este realizabilă în cadrul sistemelor de ordinul unu. Ne aflăm, iarăşi, în faţa unui exemplu tipic în care o funcţie definită pe un anumit nivel de structurare se poate

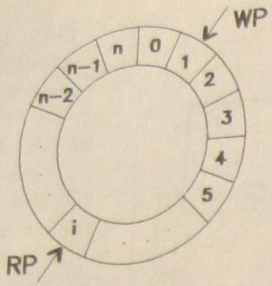



**Figura 5.27**

realiza eficient numai pe unul imediat superior. În subcapitolul 4.7 a fost definită funcţional această memorie ce este gîndită ca un şir de înregistrări cu acces la ambele capete: la unul pentru scriere iar la altul pentru citire. Într-o structură reală şirul este limitat superior, deci schema va fi centrată pe o memorie de tip *RAM* adresată prin intermediul a două numărătoare ce indică cele două capete ale şirului.

• **Structura abstractă a memoriei FIFO** este reprezentată în figura 5.27 sub forma unei reţele circulare de locaţii de memorie. Parcurgerea, adresarea, acestei reţele poate fi realizată cu doi *pointeri*, cele două numărătoare. *WP* este pointerul de scriere iar *RP* este cel de citire. La fiecare scriere se incrementează *WP* iar la fiecare citire se incrementează *RP*. Sirul înregistrat în *FIFO* este cuprins la adresele mai mari (modulo $n+1$) decît *RP* şi mai mici (modulo $n+1$) decît *WP*. Iniţial cei doi pointeri au valoare identică (şirul are lungime nulă).

O bună funcţionare presupune ca în parcurgerea cercului cei doi pointeri să nu se "depăşească" niciodată; ei se pot, cel mult, ajunge din urmă. Astfel dacă *RP* a ajuns din urmă pe *WP* atunci memoria a fost golită (*RP* a devenit, prin incrementare, egal cu *WP*). Dacă *WP* devine, prin incrementare, egal cu *RP*, atunci memoria a fost umplută la capacitatea maximă.

• **Implementarea memoriei FIFO** porneşte de la structura abstractă anterior definită şi de la reprezentarea din figura 4.41. Configuraţia propusă este reprezentată în figura 5.28 unde:

- *RAM* este elementul de stocare;

- *WN* este numărătorul asociat pointerului *WP*;

- *RN* este asociat lui *RP*;

- *MUX* permite adresarea memoriei alternativ, prin conţinutul celor două numărătoare;

- *COMP* este un comparator ce indică identitatea sau neidentitatea pointerilor;

- Ø este ciclul sistemului, obţinut prin divizarea ceasului (*CK*). Într-un ciclu complet, Ø , se poate realiza atît o scriere (în faza pentru care $Ø=1$) cît şi o citire ($Ø=0$). Latch-ul din schemă memorează care a fost ultima operaţie efectuată, în funcţie de care identitatea pointerilor semnifică *FULL* sau *EMPTY*.



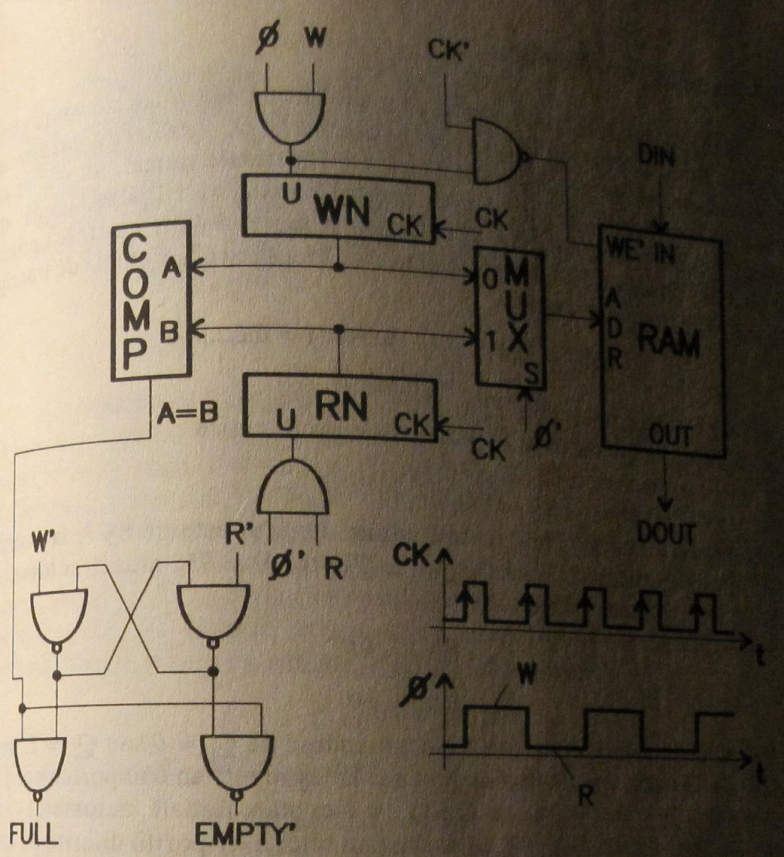



**Figura 5.28**

Semnalul *WE'* este condiţionat şi de *CK* pentru a se putea asigura timpul de set-up între stabilizarea adreselor şi comanda de scriere. În caz contrar se riscă perturbarea conţinutului unor locaţii de memorie, altele decît cea adresată.

• Numărătoarele din schemă fiind sisteme de ordinul doi, fac astfel încît întregul ansamblu aparţine la *SO2*. *Complexitatea schemei* este dictată, totuşi de complexitatea *RAM*-ului, a elementului de stocare propriuzisă. Dacă $n$ este lungimea maximă a şirului stocat în *FIFO* atunci

$$S_{FIFO}(n) = S_{RAM}(n) + S_{NUM}(n) + S_{COM+MUX}(n) + k_0 =$$
$$= O(n) + O((\log n)^2) + O(\log n) + O(1) = O(n).$$

Structura de control realizată ca *SO2* nu afectează complexitatea sistemului fiind, şi prin aceasta, justificată ca opţiune.

### 5.3.5 Numărătoare Moebius

Maniera cea mai rudimentară de a realiza un automat cu multe stări şi circuite combinaţionale minime este aceea de a folosi un *registru cu recirculare serie* şi a introduce într-un singur punct, între două bistabile numai, un circuit logic oarecare (figura 5.29). Funcţionarea acestui circuit este foarte simplă: el se comportă ca un registru de recirculare serie (paragraful 4.4.2), intrarea primului bistabil fiind "calculată" în funcţie de intrarea $X$ şi ieşirea $Q_0$. Funcţia de tranziţie a automatului este foarte simplă:

$$Q_i(t) = Q_{i+1}(t\text{-}1), \quad \text{pentru } i = 0,1,\ldots,n\text{-}1$$
$$Q_n(t) = f(Q_0(t\text{-}1), X).$$

• **Varianta cea mai simplă** o constituie cazul în care

$$Q_n(t) = Q_0(t\text{-}1).$$

Numărul maxim de stări care poate fi atunci obţinut este de $n+1$, iar numărul minim este 2. Într-adevăr dacă $Q_n\, Q_{n-1} \ldots Q_0 = 0101 \ldots 01$, atunci automatul va prezenta la ieşire numai două stări:

$$0101 \ldots 01$$

şi

$$1010 \ldots 10.$$

(Nu luăm în consideraţie starea de blocare indusă de $Q_i = 0$ sau $Q_i = 1$, pentru $i = 0,1,\ldots,n$. În acest caz circuitul nu prezintă la ieşire nici un comportament.)

Pentru $Q_i = 0$ $(i = 0,1,\ldots,n\text{-}1)$ şi $Q_n = 1$ ca stare iniţială, automatul va avea un număr de $n+1$ stări distincte, corespunzînd celor $n+1$ poziţii diferite pe care le poate ocupa unu-ul, din starea iniţială, prin deplasări succesive. Astfel

$$q_0 = 100 \ldots 0,$$
$$q_1 = 010 \ldots 0,$$
$$q_2 = 001 \ldots 0,$$
$$\vdots$$
$$q_{n+1} = 000 \ldots 1.$$

Stării $q_{n+1}$ îi va urma $q_0$ ş.a.m.d.

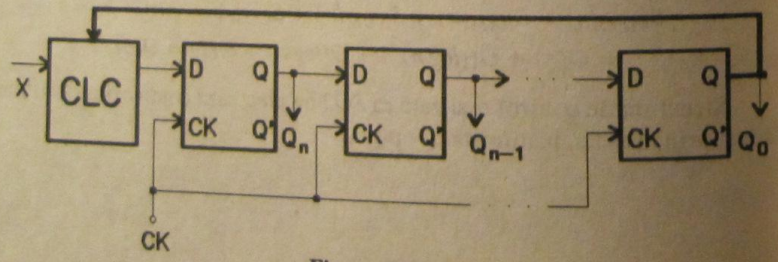



**Figura 5.29**

**Figura 5.30**

• **Numărătorul Moebius** este o variantă ce permite *dublarea numărului de stări* în condiţii de complexitate minimă a *CLC*-ului. Sugestia pentru această structură este oferită de *suprafaţa lui Moebius* reprezentată (în viziunea lui M.C.Escher) în figura 5.30. Drumul ce poate fi parcurs pe o astfel de suprafaţă este dublu faţă de lungimea benzii, deoarece este permisă parcurgerea ambelor feţe ale benzii într-un traseu continuu. Modul în care a fost construită această suprafaţă, pornind de la o bandă căreia i s-au unit capetele răsucit (oricine poate face această simplă experienţă), ne oferă soluţia pentru numărătorul Moebius. Într-adevăr va fi suficient să particularizăm funcţia $f(Q_0(t\text{-}1), X)$ sub forma

$$Q_n(t) = Q_0'(t\text{-}1).$$

Dacă starea iniţială a bistabilelor este

$$Q_n\, Q_{n-1} \ldots Q_0 = 00 \ldots 0,$$

atunci stările următoare vor fi

$$100 \ldots 0$$
$$110 \ldots 0$$
$$111 \ldots 0$$
$$\vdots$$
$$111 \ldots 1$$
$$011 \ldots 1$$
$$001 \ldots 1$$
$$000 \ldots 1$$
$$\vdots$$
$$000 \ldots 0$$

ş.a.m.d. pînă la revenirea în starea iniţială. Evident, numărul de stări ale automatului obţinut este de $2n+2$. Remarcăm faptul că *CLC*-ul asociat este practic inexistent, comportamentul automatului fiind obţinut cu un efort minimal.

• **O limită a numărătorului Moebius** (cunoscut în literatură şi ca numărător Johnson) este aceea că numărul distinct de stări oferit la ieşire cu $n$ bistabili este numai de $2n$ faţă de $2^n$ cît ar fi posibil dacă s-ar folosi capacitatea maximă de reprezentare a unei configuraţii binare de $n$ biţi.

• **Avantajele** acestor numărătoare sînt legate de facilităţile de decodificare a stărilor. Dezavantajul, anterior enunţat, permite, în compensare, o facilă identificare a stărilor. Să considerăm un numărător Moebius de patru biţi ($n=3$). Starea *1110* se individualizează prin faptul că $Q_1=1$ şi $Q_0=0$, implicit fiind valabil că $Q_3=Q_2=1$. Deci, pentru decodificarea ei este suficient circuitul cu funcţia $Q_1 \cdot Q_0'$. De asemenea, pentru a individualiza o *secvenţă de stări succesive* se foloseşte tot o poartă cu două intrări. Spre exemplu, pentru secvenţa *1000*, *1100*, *1110* poate fi folosită funcţia logică $Q_3 \cdot Q_0'$.

Al doilea avantaj este dat de faptul că la ieşirea acestor porţi *nu apar fenomene de hazard* deoarece, datorită modului în care se înlănţuie stările, niciodată nu se modifică mai mult de un bit la intrarea acestor *AND*-uri.

• **Amorsarea funcţionării acestor numărătoare** este o problemă ce se poate rezolva în două moduri. O primă metodă presupune resetarea sau setarea tuturor bistabilelor la iniţiere. A doua variantă presupune o autoamorsare printr-un *CLC* mai complex care, detectînd o configuraţie binară neconformă, să determine corectarea ei. Acest procedeu permite ca şi erori ce apar în mod aleator în funcţionare să fie corectate.

Avantajul unor scheme de autoamorsare, ce funcţionează şi ca scheme de autocorectare, este discutabil. Aceste scheme pot masca disfuncţionalităţi ce se manifestă aleator şi nu provin de la erori de structură ci de la o realizare tehnologică deficitară ce permite acţiunea nesistematică a unor surse de zgomot. Schema îşi poate redresa funcţionarea dar efectele proastei funcţionări temporare nu pot fi totdeauna evitate, ele marcînd ansamblul în care numărătorul funcţionează. Apare şi dezavantajul că "vinovatul" nu poate fi pus în evidenţă cu uşurinţă pentru că-şi maschează disfuncţionalitatea prin autocorectare. Pînă reuşim să facem o măsurătoare în sistem, numărătorul şi-a redresat comportamentul şi nu-l mai putem "prinde".

• **Aplicaţiile curente** ale numărătoarelor Moebius sînt legate de generarea unor secvenţe standard de impulsuri necesare generării contextului, relativ fix, în care funcţionează o schemă digitală.

## 5.4 Descrierea formală a automatelor finite

Am aminat descrierea formală a sistemelor ce fac obiectul acestui capitol pentru a pune în evidenţă prin exemple semnificative o serie de concepte care, dacă ar fi fost introduse pur teoretic, nu ar fi putut fi prea uşor asimilate. Observăm faptul că ieşirea unui automat nu se comportă univoc determinată de semnalele aplicate pe intrare. Evoluţia ieşirii lui este condiţionată şi de *starea* sa internă. Cînd unui



numărător îi dăm comanda *Up* nu ştim exact care va fi valoarea ieşirii lui decît dacă ştim şi care este valoarea actuală a acesteia. Starea internă a unui automat, sau mai simplu spus starea automatului, determină evoluţia sa împreună cu intrarea automatului. În aceasta constă autonomia asociată sistemelor de ordinul doi. Exemple particulare de *SO2* am dat în subcapitolele anterioare, vom da şi altele în continuare, dar, pentru ceea ce urmează, vom porni de la un fundament teoretic general.

### 5.4.1 Definiţii. Proprietăţi

**Definiţia 5.1** Un *automat de tip Mealy* este definit prin cvintuplul

$$A = (X, Y, Q, \lambda, \delta),$$

unde entităţile ce-l compun au următoarea semnificaţie:

$X$ - este mulţimea configuraţiilor binare de *intrare*,

$Y$ - este mulţimea configuraţiilor binare de *ieşire*,

$Q$ - este mulţimea configuraţiilor binare de *stare*,

$\lambda$ - este funcţia de tranziţie a stării, de forma

$$\lambda : X \times Q \to P^*(Q),$$

$\delta$ - este funcţia de tranziţie a ieşirii, de forma

$$\delta : X \times Q \to P^*(Y).$$

(Reamintim că notăm prin $P^*(A)$, mulţimea nevidă a părţilor mulţimii $A$.) □

**Definiţia 5.2** Un *automat de tip Moore* este derfinit prin cvintuplul

$$A = (X, Y, Q, \lambda, \eta),$$

unde primele patru entităţi au fost descrise în *Definiţia 5.1* iar

$\eta$ - este funcţia de tranziţie a ieşirii, de forma

$$\eta : Q \to P^*(Y). \square$$

Diferenţa între cele două automate este dată de faptul că la cel de tip Moore ieşirea se deduce strict din stare, efectul intrării asupra ieşirii fiind dat numai prin intermediul stării. Unui anumit element din $X \times Q$ sau din $Q$ îi pot corespunde mai multe elemente din $Q$ sau $Y$, fapt ce a determinat folosirea mulţimii părţilor acestor ultime două mulţimi, în definiţiile anterioare. În cazul general tranziţia unui automat putînd fi, astfel, şi nedeterministă.

**Definiţia 5.3** Dacă orice element din $P^*(Q)$ şi $P^*(Y)$ conţine numai un element (are cardinalul unu), atunci automatul este *determinist*. În caz contrar el este nedeterminist. Funcţiile ce apar în definiţiile anterioare se rescriu după cum urmează:

$$\lambda : X \times Q \to Q,$$
$$\delta : X \times Q \to Y,$$
$$\eta : Q \to Y. \square$$

**Definiţia 5.4** Dacă mulţimile $X$, $Y$, $Q$ sînt finite atunci automatul este finit. □

Automatele cu stări finite au fost introduse pentru a pune în evidenţă un efect mai subtil decît aparenţele lasă să se întrevadă. În esenţă, un automat finit este unul ce *are un număr de stări independent de lungimea şirurilor de configuraţii binare aplicate la intrare*. Oricît ar fi de lung şirul, chiar infinit, el poate fi tratat cu un automat cu un număr finit de stări.

În cele ce urmează vom folosi numai automate deterministe şi finite, cu excepţia cazurilor cînd vom specifica, eventual, altfel.

**Definiţia 5.5** Dacă pentru orice $q_j \in Q$ şi $x_i \in X$

$$\lambda(x_i, q_j) = q_k$$

cu $q_k \in Q$, atunci automatul este *neiniţial*. Dacă $q_k$ ia valoare numai în $Q_1 \subset Q$ atunci automatul este *iniţial*, iar mulţimea $Q_0$, astfel definită încît $Q_0 \cup Q_1 = Q$ şi $Q_0 \cap Q_1 = \varnothing$, este mulţimea stărilor iniţiale. □

**Definiţia 5.6** Dacă $|Q_0| = 1$ automatul este strict iniţial. □

Stările iniţiale nu provin dintr-o evoluţie a automatului. De regulă, spunem că automatul este adus printr-o comandă exterioară (diferită de semnalele de pe intrarea $X$) într-o stare iniţială.

**Definiţia 5.7** O stare a unui automat iniţial este considerată *inaccesibilă* dacă orice traiectorie, în spaţiul stărilor, ce porneşte din stări iniţiale, nu o conţine. □

Putem vorbi de stări inaccesibile numai în cazul automatelor iniţiale.

**Definiţia 5.8** Două stări $q_i$ şi $q_j$ sînt echivalente dacă evoluţia automatului, pornind din aceste stări, generează la ieşire şiruri de configuraţii binare identice. □

**Definiţia 5.9** Un *semiautomat* este definit prin tripletul

$$SA = (X, Q, \lambda),$$

unde entităţile ce-l compun sînt descrise în *Definiţia 5.1*. □

Un semiautomat este finit şi determinist în sensul în care şi automatele au aceste proprietăţi. Conceptul de semiautomat separă acea parte a automatului responsabilă de evoluţia internă, de autonomia propriuzisă. Vom vedea că optimizarea unui automat se reduce, în esenţă, la optimizarea semiautomatului asociat. Dacă dintr-un automat extragem circuitele asociate ieşirii obţinem semiautomatul asociat.

**Definiţia 5.10** Un automat este *imediat* dacă

$$\delta(x_i(t), q_j(t)) = y_k(t)$$

pentru un automat de tip Mealy şi

$$\eta(q_j(t)) = y_k(t)$$



pentru un automat de tip Moore. Prin $t$ am notat variabila discretă timp. □

Deci, la un automat imediat, *ieşirea depinde de starea curentă*.

**Definiţia 5.11** Un automat este *cu întîrziere* dacă

$$\delta(x_i(t), q_j(t)) = y_k(t+1)$$

pentru un automat de tip Mealy şi

$$\eta(q_j(t)) = y_k(t+1)$$

pentru un automat de tip Moore. □

În acest al doilea caz *ieşirea depinde de starea din ciclul anterior* de ceas. Pentru un semiautomat avem, evident:

$$q_k(t) = \lambda(x_i(t-1), q_j(t-1)).$$

**Propoziţia 5.1** Reacţia ieşirii unui automat faţă de variaţia intrării este diferenţiată după cum urmează:

- automatul Mealy imediat urmăreşte direct prin ieşire variaţia intrării;

- automatul Mealy cu întîrziere şi Moore imediat urmăresc variaţiile intrării cu întîrziere de un tact;

- automatul Moore cu întîrziere sesizează la ieşire variaţiile intrării cu întîrziere de două tacturi. □

*Demonstraţia* propoziţiei anterioare este imediată, după cum urmează:

- $y_k(t) = \delta(x_i(t), q_j(t))$ pentru automatul de tip Mealy imediat;

- $y_k(t) = \delta(x_i(t-1), q_j(t-1))$ pentru automatul de tip Mealy cu întîrziere şi

- $y_k(t) = \eta(q_j(t)) = \eta(\lambda(x_i(t-1), q_i(t-1))$ pentru automatul de tip Moore imediat;

- $y_k(t) = \eta(q_j(t-1)) = \eta(\lambda(x_i(t-2), q_i(t-2))$ pentru automatul de tip Moore cu întîrziere. □

• **Structurarea automatelor finite** se poate realiza simplu, utilizînd *CLC*-uri şi registre (comandate pe front, deci realizate cu bistabili de tip master-slave). Cele

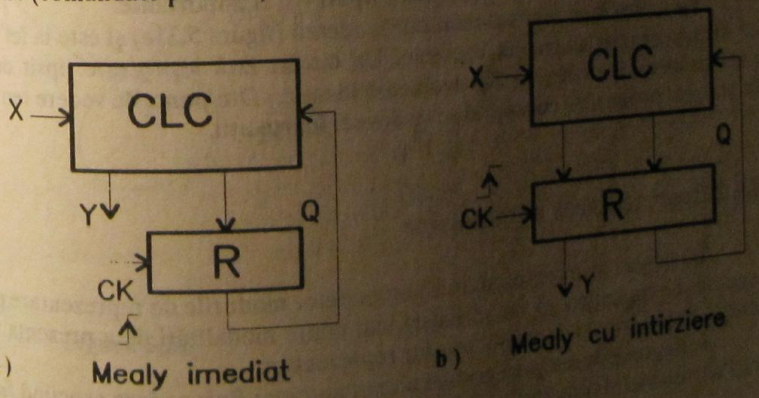
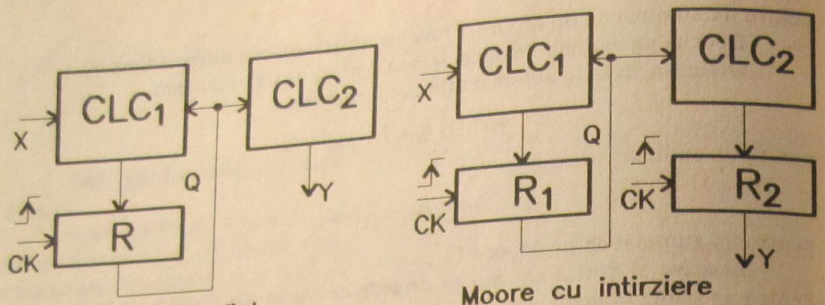



**Figura 5.31**



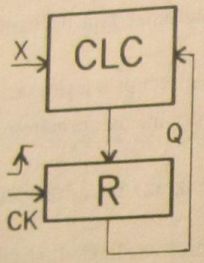




**Figura 5.31 (continuare)**

patru tipuri de automate, anterior evidenţiate, sînt reprezentate în figura 5.31. Rezultă încă o dată modul în care aceste tipuri de automate reacţionează la variaţiile semnalului aplicat intrării. Toate cele patru tipuri de automate sînt semnificative deoarece, în funcţie de aplicaţie, ele se integrează diferit, ca modalitate de a reacţiona, într-o schemă concretă. La nivelul abordării de sistem, preliminare, distincţiile între cele patru tipuri vor fi importante.

Structura unui semiautomat este evidentă (figura 5.31*e*) şi este la fel de clar că nu corespunde niciunèia concrete. Un circuit fără ieşire este lipsit de sens. (Acceptăm cel mult că $Q = Y$ şi evoluează identic.) Din punct de vedere teoretic şi în practica proiectării conceptul se va dovedi foarte util.

### 5.4.2 Reprezentarea automatelor

Înţelegem prin reprezentarea automatelor modurile de reprezentare pentru funcţiile de tranziţie $\lambda$, $\delta$ sau $\eta$. Există mai multe modalităţi de a prezenta aceste funcţii, ţinînd cont şi de scopul acestor reprezentări.

• **Reprezentarea prin grafuri** a unui automat finit se face asociind fiecărei stări cîte un *nod* şi tranziţiilor *arce*. Arcele vor fi orientate şi marcate cu condiţia ce



determină tranziţia, atunci cînd aceasta este condiţionată. Ieşirea este marcată distinct la automatele de tip Mealy şi cele de tip Moore. La primele ieşirea este marcată pe arc iar la ultimele este asociată strict nodului, în conformitate cu definiţiile celor două tipuri de automate.

**Exemplul 5.3** Vom folosi pentru exemplificările din acest paragraf o pereche de automate echivalente, unul de tip Mealy şi unul de tip Moore.

Figura 5.32 reprezintă un automat de tip Mealy cu trei stări ($q_0$, $q_1$ şi $q_2$). Configuraţiile binare acceptate la intrare sînt $x_1$ şi $x_2$. Mulţimea $Y$ are patru elemente: $y_0$, $y_1$, $y_2$ şi $y_3$. Dacă automatul se află în starea $q_0$, va trece în $q_1$ generînd pe ieşire $y_1$ dacă pe intrare nu este $x_1$ sau va trece în $q_2$ generînd pe ieşire $y_2$ dacă pe intrare este $x_1$. Din starea $q_1$, în care generează pe ieşire $y_1$, automatul trece necondiţionat în starea $q_0$. Restul grafului se interpretează similar.

Automatul Moore cu un comportament la borne identic este reprezentat prin graful din figura 5.33. Conform definiţiei, fiecare stare are asociată o configuraţie binară de ieşire. Mulţimea stărilor automatului este $Q = \{q_4, q_5, q_6, q_7, q_8\}$. Mulţimea $Y$ este, evident, aceeaşi. La fel în cazul mulţimii $X$. În starea $q_4$ ieşirea este $y_1$, la fel în $q_5$. Iar în starea $q_6$ ieşirea este $y_2$ ş.a.m.d. Funcţia de tranziţie a stării depinde de $X = \{x_1, x_2\}$. Din starea $q_6$, spre exemplu, automatul comută în $q_7$ sau $q_8$ în funcţie de inexistenţa pe intrare a configuraţiei binare $x_2$ sau de existenţa ei. Similar pot fi interpretate celelalte tranziţii. □

Se poate uşor observa că cele două automate au un comportament la borne identic. Numărul de stări este de regulă mai mare la un automat Moore decît la unul

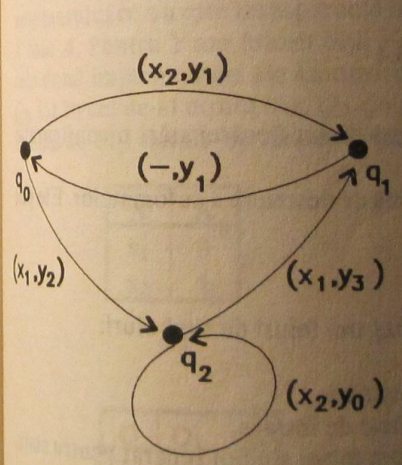



**Figura 5.32**

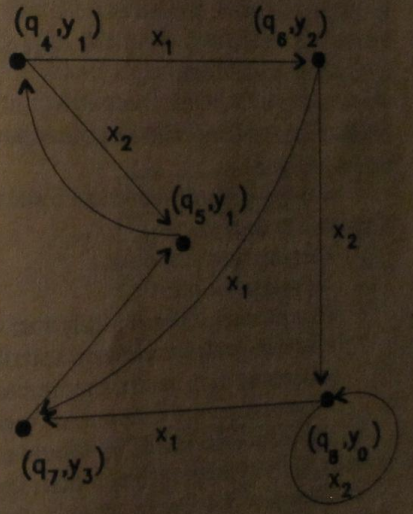



**Figura 5.33**

Mealy deoarece stările "memorează", în primul caz, şi efectul modificării configuraţiilor binare aplicate intrării.

• **Tabelele de tranziţie** reprezintă tot o formă primară de descriere a funcţionării unui automat. În tabelele 5.4 sînt reprezentate pe rînd funcţiile de tranziţie ale automatului Mealy şi cele ale automatului Moore echivalent. Tabelul 5.4*a* reprezintă funcţia $\lambda$ ce determină starea următoare a automatului pornind de la cea curentă şi de la simbolul aplicat pe intrare. Astfel, din starea $q_2$, dacă pe intrare se aplică $x_1$, atunci starea următoare va fi $q_1$ ş.a.m.d. La fel pot fi citite şi

| Q / X | $q_0$ | $q_1$ | $q_2$ |
|---|---|---|---|
| $x_1$ | $q_2$ | $q_0$ | $q_1$ |
| $x_2$ | $q_1$ | $q_0$ | $q_2$ |

a)

| Q / X | $q_0$ | $q_1$ | $q_2$ |
|---|---|---|---|
| $x_1$ | $y_2$ | $y_1$ | $y_3$ |
| $x_2$ | $y_1$ | $y_1$ | $y_0$ |

b)

| Q / X | $q_4$ | $q_5$ | $q_6$ | $q_7$ | $q_8$ |
|---|---|---|---|---|---|
| $x_1$ | $q_6$ | $q_4$ | $q_7$ | $q_5$ | $q_7$ |
| $x_2$ | $q_5$ | $q_4$ | $q_8$ | $q_5$ | $q_8$ |

c)

| Q | $q_4$ | $q_5$ | $q_6$ | $q_7$ | $q_8$ |
|---|---|---|---|---|---|
| Y | $y_1$ | $y_1$ | $y_2$ | $y_3$ | $y_0$ |

d)

**Tabelul 5.4**

celelalte tabele. Observăm că ultimul asociază direct fiecărei stări o configuraţie binară de ieşire.

• **Organigramele** reprezintă o altă formă de descriere a automatelor. Ele pot fi realizate în două forme:

- organigrame simbolice,
- organigrame binare.

În cazul unui automat Mealy vom utiliza trei feluri de simboluri:

- cercuri, pentru a reprezenta stările;
- dreptunghiuri, pentru a reprezenta ieşirile;
- romburi, pentru a reprezenta variabilele de intrare.

Pentru automatele de tip Moore nu vom folosi simbol separat pentru stare, notînd alături de dreptunghiul asociat ieşirii numele stării.

**Exemplul 5.4** Organigramele simbolice asociate automatelor definite în *Exemplul 5.3* sînt reprezentate în figura 5.34 şi în figura 5.35. □

**Exemplul 5.5** Pentru a reprezenta *organigramele binare* ce descriu cele două automate este necesară stabilirea codurilor binare asociate elementelor celor trei mulţimi ce definesc automatul. Tabelele 5.5 definesc această codificare binară. Într-un exemplu concret, mulţimile $X$ şi $Y$ sînt codificate de cel ce defineşte



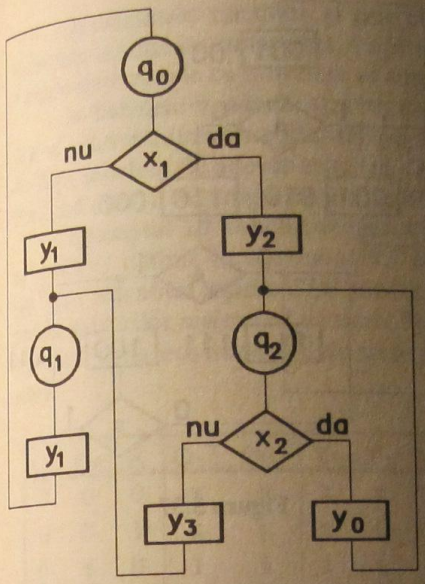



**Figura 5.34**

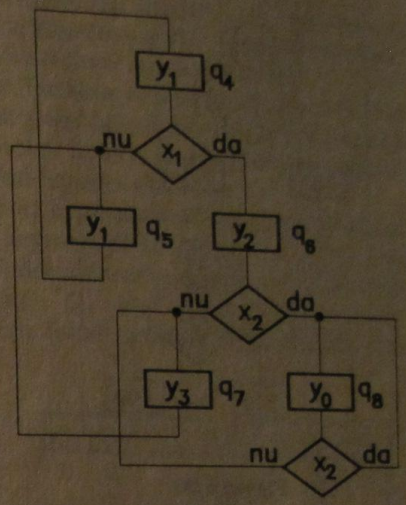



**Figura 5.35**

automatul, ele depinzînd de contextul în care funcţionează circuitul. Mulţimea $Q$ poate fi definită de către proiectant deoarece depinde de structura internă a automatului ce nu este transparentă la borne. Bitul ce codifică elementele mulţimii $X$ este $A$. Pentru $Y$ am folosit biţii $Y_2$, $Y_1$ şi $Y_0$. Mulţimea $Q$ este distinctă pentru cele două implementări ale automatului. În primul caz putem folosi doi biţi, $Q_1$ şi $Q_0$, iar în cel de al doilea caz, $Q_2$, $Q_1$ şi $Q_0$. Codificarea stărilor se face cu numărul minim de biţi admis de dimensiunea mulţimii $Q$. În figurile 5.36 şi 5.37 au fost

| X | A |
|---|---|
| $x_1$ | 0 |
| $x_2$ | 1 |

a)

| Y | $Y_2$ | $Y_1$ | $Y_0$ |
|---|---|---|---|
| $y_0$ | 1 | 0 | 0 |
| $y_1$ | 0 | 0 | 1 |
| $y_2$ | 1 | 1 | 0 |
| $y_3$ | 1 | 1 | 1 |

b)

| Q | $Q_1$ | $Q_0$ |
|---|---|---|
| $q_0$ | 0 | 1 |
| $q_1$ | 1 | 0 |
| $q_2$ | 1 | 1 |

c)

| Q | $Q_2$ | $Q_1$ | $Q_0$ |
|---|---|---|---|
| $q_4$ | 0 | 0 | 1 |
| $q_5$ | 0 | 1 | 0 |
| $q_6$ | 0 | 0 | 0 |
| $q_7$ | 1 | 1 | 1 |
| $q_8$ | 0 | 1 | 1 |

d)

**Tabelul 5.5**

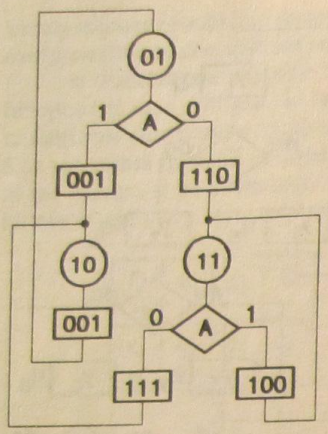



Figura 5.36

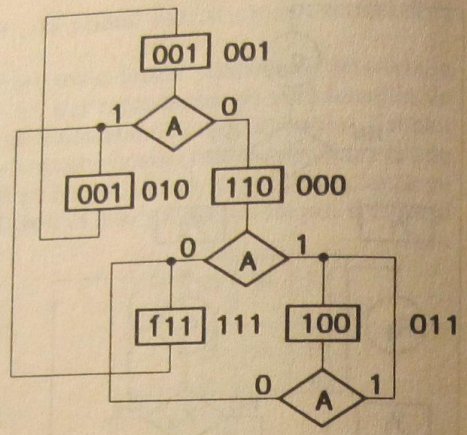



Figura 5.37

redesenate organigramele în varianta binară. Această reprezentare este mai apropiată de una utilă în proiectarea acestor automate. □

**Exemplul 5.6** O situaţie mai complexă este reprezentată în fragmentul de organigramă din figura 5.38, asociată unui automat pentru care $X = \{x_0, x_1, x_2\}$, codificată conform tabelului 5.6. Rezultă fragmentul de organigramă binară din

| X | $X_1$ | $X_0$ |
|---|---|---|
| $x_1$ | 0 | 0 |
| $x_2$ | 0 | 1 |
| $x_3$ | 1 | 1 |

Tabelul 5.6

figura 5.39. Complexitatea organigramei nu creşte, fiind vorba numai de o reconfigurare. □

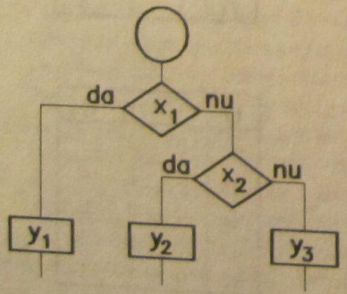



Figura 5.38

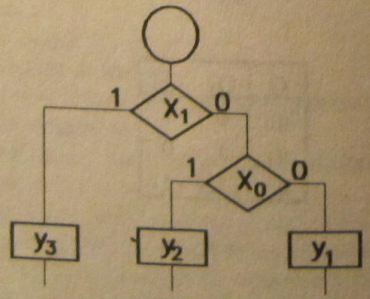



Figura 5.39



Automatele imediate şi cele cu întîrziere sînt descrise prin organigrame identice; rezultă, în consecinţă, un proces de proiectare similar în cele două cazuri. Pentru automatele cu întîrziere se adaugă un registru variantei imediate.

• **Tabelele de adevăr** sînt reprezentări secunde ce rezultă, de regulă, dintr-o formă primară, spre exemplu, dintr-o organigramă binară. Aceste tabele reprezintă de fapt tabelele de adevăr asociate *CLC*-urilor ce intră în structura automatului. Pornind de la o organigramă ele pot fi direct reprezentate.

**Exemplul 5.7** Tabelul de tranziţie asociat automatului Mealy (imediat sau cu întîrziere) definit în *Exemplul 5.3* este reprezentat în tabelul 5.7, unde, prin $Q_1^+Q_0^+$ am notat ieşirile *CLC*-ului ce calculează starea următoare. Aceste ieşiri se aplică intrărilor *registrului de stare*. Pentru $Q_1^+Q_0^+ = 00$ nu am definit funcţia de tranziţie deoarece acest cod de stare nu este folosit, automatul neavînd decît trei stări.

| $Q_1$ | $Q_0$ | A | $Q_1^+$ | $Q0^+$ | $Y_2$ | $Y_1$ | $Y_0$ |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | x | x | x | x | x |
| 0 | 0 | 1 | x | x | x | x | x |
| 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |

Tabelul 5.7

| $Q_1$ | $Q_0$ | $Q_1^+$ | $Q0^+$ | $Y_2$ | $Y_1$ | $Y_0$ |
|---|---|---|---|---|---|---|
| 0 | 0 | x | x | x | x | x |
| 0 | 1 | 1 | A' | A' | A' | A |
| 1 | 0 | 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | A | 1 | A' | A' |

Tabelul 5.8

O variantă posibilă pentru tabelul de adevăr este cea în care variabilele ce descriu intrarea automatului să fie incluse în coloanele din dreapta tabelului. Dăm această variantă în tabelul 5.8. □

**Exemplul 5.8** Tabelul de tranziţie, cu variabile incluse în coloanele din dreapta tabelului, pentru automatul Moore anterior definit este reprezentat în tabelul 5.9. Starea *000* este urmată de *111* dacă *A=0* sau *011* dacă *A=1*, deci succesorul poate fi notat prin *A'11* în tabelul de tranziţie. La fel au fost completate

| $Q_2$ | $Q_1$ | $Q_0$ | $Q_2^+$ | $Q1^+$ | $Q_0^+$ | $Y_2$ | $Y_1$ | $Y_0$ |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | A' | 1 | 1 | 1 | 1 | 0 |
| 0 | 0 | 1 | 0 | A | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | A' | 1 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | x | x | x | .x | x | x |
| 1 | 0 | 1 | x | x | x | x | x | x |
| 1 | 1 | 0 | x | x | x | x | x | x |
| 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |

Tabelul 5.9

şi următoarele rînduri ce definesc tranziţia stării. Tranziţia ieşirii se completează simplu cu codurile de ieşire asociate direct fiecărei stări, conform organigramei din figura 5.37. □

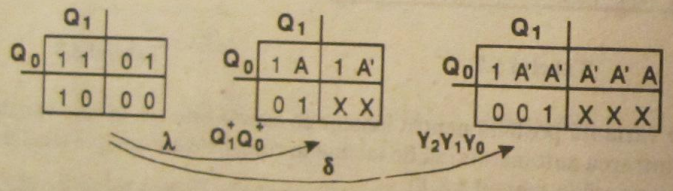
• **Diagramele de tranziţie** corespund tabelelor de adevăr oferind o reprezentare mai compactă, ce oferă, în plus, şi posibilitatea minimizării *CLC*-ului asociat automatului.

**Exemplul 5.9** În figura 5.40 sînt date diagramele Veich-Karnaugh asociate automatului Mealy folosit ca exemplu. În figura 5.40*a* sînt definite funcţiile $\lambda$ şi $\delta$ în funcţie de o diagramă de referinţă (cea de la care pornesc săgeţile). Aceste diagrame sînt detaliate în figura 5.40*b*, pentru a se putea extrage eficient funcţiile logice ce rezultă sub forma următoare:

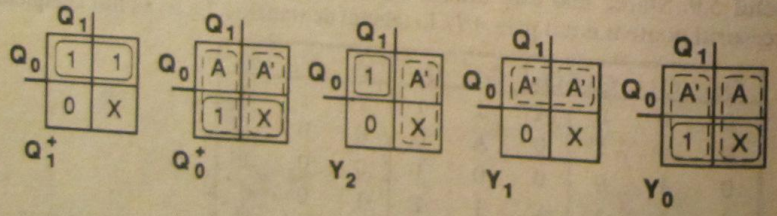
$$Q_1^+ = Q_0$$
$$Q_0^+ = Q'_0 + Q_1 A + Q'_1 A'$$
$$Y_2 = Q_1 Q_0 + Q'_1 A'$$
$$Y_1 = Q_0 A'$$
$$Y_0 = Q'_0 + Q_1 A' + Q'_1 A\,.\ \square$$

**Exemplul 5.10** Diagramele de tranziţie asociate variantei de tip Moore sînt reprezentate în figura 5.41. Din ele rezultă:

$$Q_2^+ = Q'_2 Q_1 Q_0 A' + Q'_1 Q'_0 A'$$
$$Q_1^+ = Q_1 Q_0 + Q'_1 Q'_0 + Q_0 A$$



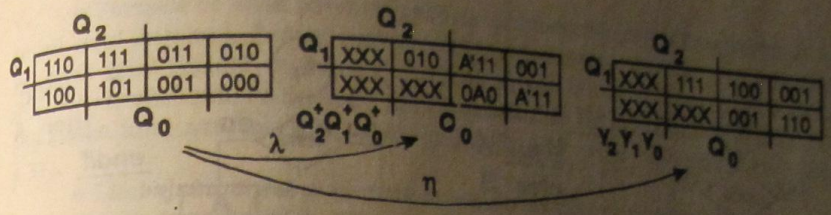

a)



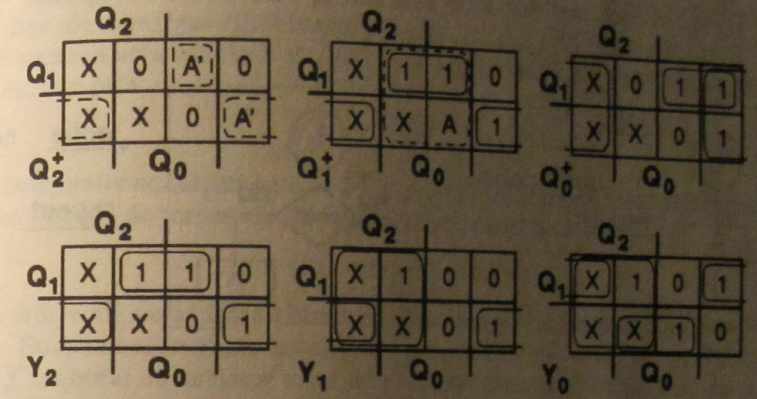

b)

**Figura 5.40**






a)




b)

**Figura 5.41**

$$Q_0^+ = Q'_0 + Q'_2 Q_1$$
$$Y_2 = Q_1 Q_0 + Q'_1 Q'_0$$
$$Y_1 = Q_2 + Q'_1 Q'_0$$
$$Y_0 = Q_2 + Q_1 Q'_0 + Q'_1 Q_0\,.\ \square$$

• **Pseudocodul** reprezintă o altă modalitate de descriere a funcţiilor de tranziţie ale unui automat. Folosirea lui dă o descriere *structurată* şi în consecinţă restrictivă pentru funcţionarea unui automat. Oricărei proceduri scrise în pseudocod i se poate asocia o organigramă. *Reciproca nu este* însă *adevărată*.

Mecanismele de structurare descrise în pseudocod sînt *restrictive*, dar oferă avantajul controlării acurateţei la complexităţi mari. În figura 5.42 este reprezentată corespondenţa dintre modurile de structurare ale unei organigrame şi ale unei proceduri. Observăm că fiecărei structuri în pseudocod îi corespunde o suborganigramă cu o singură "intrare" şi o singură "ieşire". O suborganigramă de tipul celor din figura 5.36 nu poate fi direct descrisă în pseudocod. Într-adevăr, tranziţia din starea *11*, pentru *A=0*, în starea *10* nu poate fi formalizată în pseudocod, dacă evoluţia în stările *01* şi *10* se constituie într-o buclă *while* iniţiată în *01*.

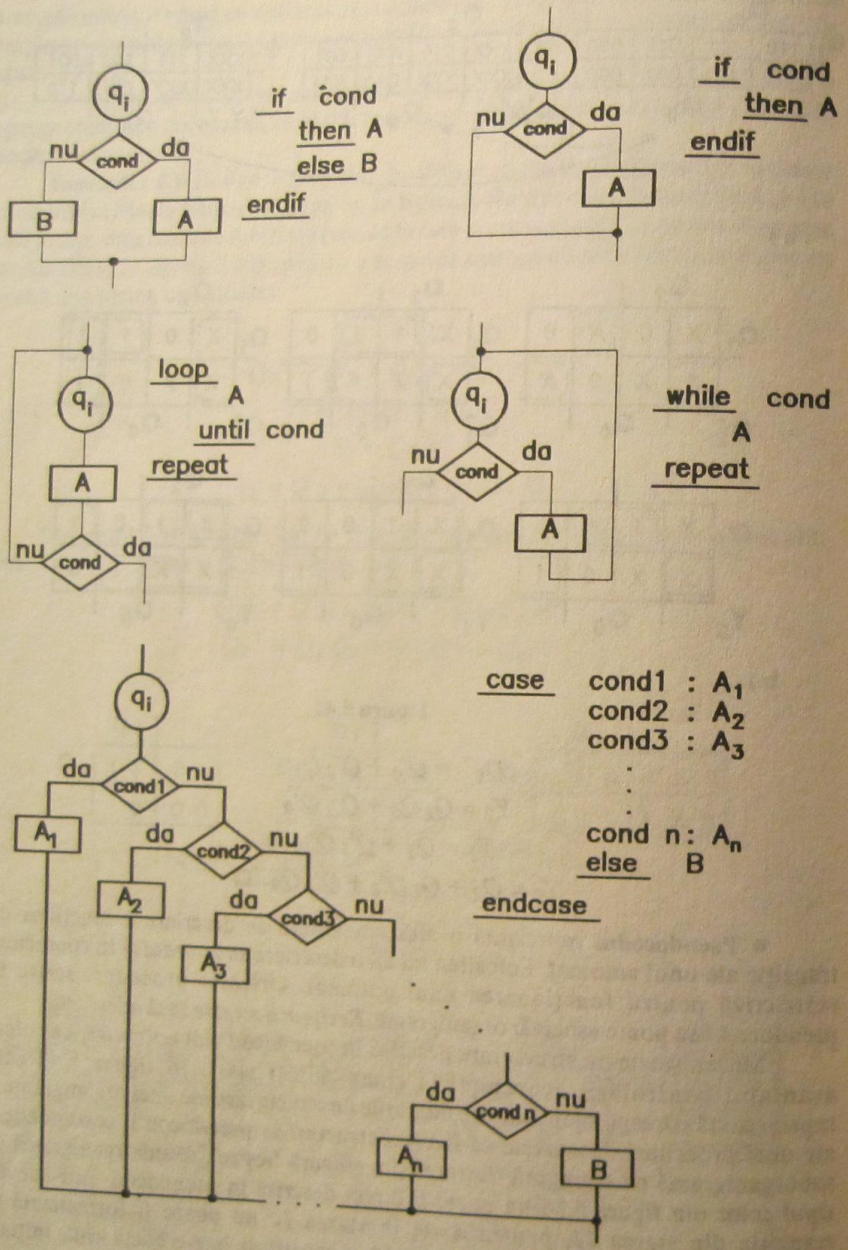



Figura 5.42



• **Formele de undă** pot oferi uneori o descriere sugestivă comportamentului ieşirii unui automat. Pentru automatele preponderent generatoare (cu evoluţie minimal condiţionată de intrare) această formă de reprezentare se dovedeşte deosebit de eficientă. Proiectarea unui generator de ciclu pentru o memorie *RAM*, sub forma unui automat finit, poate porni de la o astfel de reprezentare (figura 4.31).

• **Limbajul natural** se impune uneori în descrierea automatelor finite cu un număr foarte mare de stări. (Ne amintim că şi în cazul *CLC*-urilor cu un număr mare de intrări am fost puşi într-o situaţie similară.)

**Exemplul 5.11** Se cere proiectarea unui automat finit la care $X$ este o mulţime de configuraţii binare $\{X_0, X_1, ..., X_{n-1}\}$-în care fiecare $X_i$ este reprezentarea unui număr $N \in [0, 2^m-1]$. Mulţimile $Q$ şi $Y$ sînt identice, $Q \equiv Y$, şi sînt definite prin

$$Y = \{Y_0, Y_1, ..., Y_{p-1}\}.$$

Sînt îndeplinite condiţiile ca $n = 2^m$ şi $p = 2^q$ unde $m$ şi $q$ sînt întregi pozitivi şi $q > m$. Funcţia de tranziţie a automatului este astfel încît

$$Q(t) = Q(t\text{-}1) + X(t\text{-}1)$$

unde prin + am notat suma aritmetică. Automatul este iniţial în $Y_0$.

Schema automatului anterior descris este reprezentată în figura 5.43 unde prin $\sum$ am notat un sumator binar de $q$ biţi. Automatul va suma şiruri de cel mult $2^{q-m}$ numere, reprezentate pe $m$ biţi, pornind din starea $Y_0=Q_0$ în care ajunge prin comanda forţată *CLR*. Altfel spus este realizată o sumare *modulo* $2^q$. □

Este evident că folosirea oricărui mod de reprezentare anterior enunţat este practic imposibilă chiar pentru un $q$ rezonabil. De asemenea observăm că *CLC*-ul folosit este unul standard definibil, indiferent de dimensiune, printr-o regulă simplă şi clară.

**Exemplul 5.12** Vom porni de această dată de la o schemă, figura 5.44, pe care o vom explica. Pentru $I=0$ se încarcă în registrul de stare $R$ o valoare din $X$ (cea care se află pe intrare în ciclul de ceas în care $I$ are această valoare). Pe ieşire va apare un cod ce indică poziţia binară cea mai semnificativă cu valoare *1* din $R$. Bitul indicat va fi şters cu primul front al *CK*, ce urmează, ieşirea modificîndu-se conform situaţiei nou create ş.a.m.d. (Un exerciţiu util pentru cititor ar fi acela de a desena concret, cu circuite de catalog, un automat pentru $n=24$ (este preferată o soluţie ce nu conţine circuite inversoare)). □

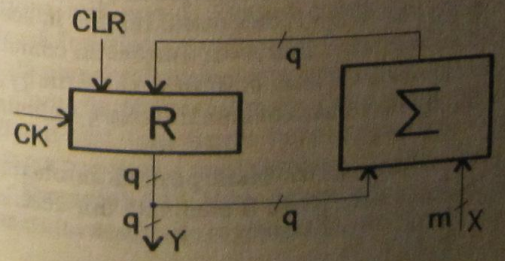



Figura 5.43

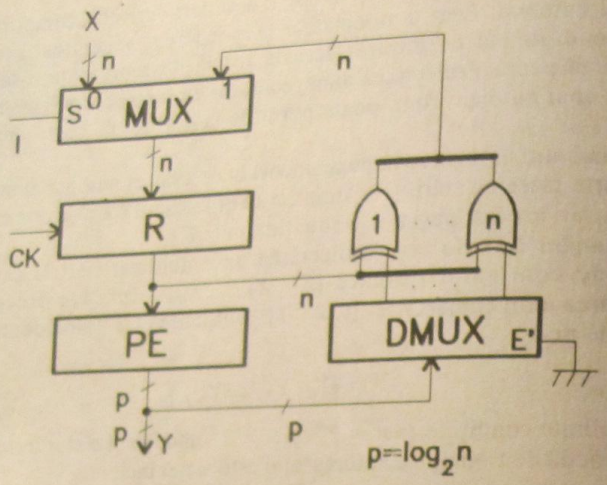



**Figura 5.44**

Automatul cu $2^n$ stări descris anterior nu poate fi definit eficient prin procedeele formale anterior expuse.

### 5.4.3 Minimizarea automatelor

Prin minimizarea unui automat înţelegem definirea unui automat echivalent, cu un număr mai mic de stări. Acest lucru este posibil prin evidenţierea submulţimilor stărilor echivalente şi înlocuirea stărilor echivalente numai cu una din ele. Comportamentul la borne rămîne identic optimizîndu-se structura internă a automatului finit. În cazul automatelor iniţiale vor fi înlăturate şi stările inaccesibile.

Detectarea *stărilor echivalente* (*Definiţia 5.8*) şi a celor *inaccesibile* (*Definiţia 5.7*) reprezintă operaţiile cele mai importante în procesul de minimizare.

**Exemplul 5.13** Considerăm automatul Moore, strict iniţial în $q_1$, descris prin tabelul 5.10. Observăm că în starea $q_3$, pentru $A=1$, automatul rămîne în aceeaşi stare. La fel se comportă în $q_5$. Din ambele stări, dacă $A=0$, automatul comută în $q_4$. Atît în $q_3$ cît şi în $q_5$ ieşirea automatului ia valoarea $0$. Rezultă că stările $q_3$ şi $q_5$ sînt echivalente. Vom menţine în definiţia automatului numai starea $q_3$, eliminînd starea $q_5$.

Starea $q_1$, fiind unica stare iniţială, prin nici o evoluţie posibilă automatul nu poate ajunge în $q_6$. Rezultă că şi această stare poate fi eliminată din descrierea automatului.



| Q / X | $q_1$ | $q_2$ | $q_3$ | $q_4$ | $q_5$ | $q_6$ |
|---|---|---|---|---|---|---|
| A | $q_2$ | $q_3$ | $q_3$ | $q_1$ | $q_5$ | $q_4$ |
| A' | $q_2$ | $q_5$ | $q_4$ | $q_1$ | $q_4$ | $q_6$ |
| Y | 1 | 0 | 0 | 1 | 0 | 1 |

**Tabelul 5.10**

| Q / X | $q_1$ | $q_2$ | $q_3$ | $q_4$ |
|---|---|---|---|---|
| A | $q_2$ | $q_3$ | $q_3$ | $q_1$ |
| A' | $q_2$ | $q_5$ | $q_4$ | $q_1$ |
| Y | 1 | 0 | 0 | 1 |

**Tabelul 5.11**

În urma celor două operaţii se obţine automatul descris prin tabelul 5.11. Numărul de stări a fost redus de la *6* la *4* cu avantaje date de posibilitatea realizării unui *CLC* mai simplu şi a unui registru cu o celulă de memorare în minus. □

Numai rareori astfel de procedee se impun deoarece în majoritatea cazurilor reuşim să definim din start într-o manieră eficientă automatele pe care le utilizăm. În cazul unor structuri foarte complexe se pune problema minimizării, caz în care, de regulă, se folosesc procedee automate de minimizare.

## 5.5 Implementarea automatelor finite

Problema fundamentală ce se pune în realizarea concretă a automatelor finite este aceea a *optimizării automatului ca semiautomat*. Inginerul proiectant are maximă libertate de mişcare în realizarea funcţiei de tranziţie a stării. Mulţimea stărilor, $Q$, reprezintă spaţiul de manevră al celui ce construieşte un automat. Efecte importante de minimizare globală sau de realizabilitate în condiţii concrete dificile se pot obţine printr-o convenabilă *codificare binară a stărilor* automatului.

### 5.5.1 Codificarea stărilor

Codificarea elementelor mulţimilor $X$ şi $Y$ este strict determinată prin definiţia iniţială a automatului. Aceste coduri sînt impuse de contextul în care va funcţiona automatul. Efectul codificării mulţimii $Q$ într-un mod sau altul nu afectează funcţionarea circuitului la nivelul definiţiei abstracte. În acest caz este afectată numai mărimea (complexitatea) structurii care implementează automatul finit. Codurile elementelor din $Q$ nu sînt niciodată accesibile la borne. Efectele codificării sînt însă foarte importante, după cum vom vedea în continuare, în realizarea concretă a unui automat.

**Exemplul 5.14** Vom reface calculul relaţiilor ce definesc *CLC*-ul asociat automatului descris de organigrama din figura 5.36, modificînd numai codul asociat

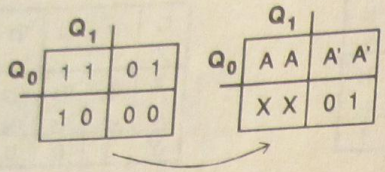



Figura 5.45

stării $q_1$ care devine *00* din *10*, cît a fost alocat în *Exemplul 5.9*. În figura 5.45 sînt reprezentate diagramele de tranziţie a stării, din care rezultă

$$Q_1^+ = Q'_1 Q_0 A' + Q_1 A$$
$$Q_0^+ = Q'_0 + Q_1 A + Q'_1 A'.$$

Funcţiile logice care definesc tranziţia stării au rezultat mult mai complexe decît în prima variantă de codificare. □

În funcţie de codificarea aleasă se poate obţine o structură mai simplă sau mai puţin simplă. Există o regulă simplă după care se poate determina codificarea asociată implementării minimale? Nu se poate concepe un algoritm simplu pentru cazul general, dar se pot formula nişte orientări practice deosebit de utile.

Codificarea stărilor nu are implicaţii numai asupra minimizării structurale a automatelor. Vom studia două variante extreme de codificare pentru a le putea folosi în procesul de alocare a codurilor binare elementelor din $Q$. Libertatea pe care o avem datorită izomorfismului permis în spaţiul stărilor o putem folosi pentru a obţine adaptarea funcţionării automatului la anumite condiţii restrictive.

• **Codificarea stărilor conform principiului dependenţei reduse** în funcţie de o variabilă, presupune ca stările care urmează unei stări date să fie astfel codificate încît un singur bit al codului să depindă de această variabilă.

**Exemplul 5.15** Fragmentul de organigramă reprezentat în figura 5.46*a* are stările codificate cu dependenţă redusă faţă de bitul de intrare $B$. Într-adevăr,

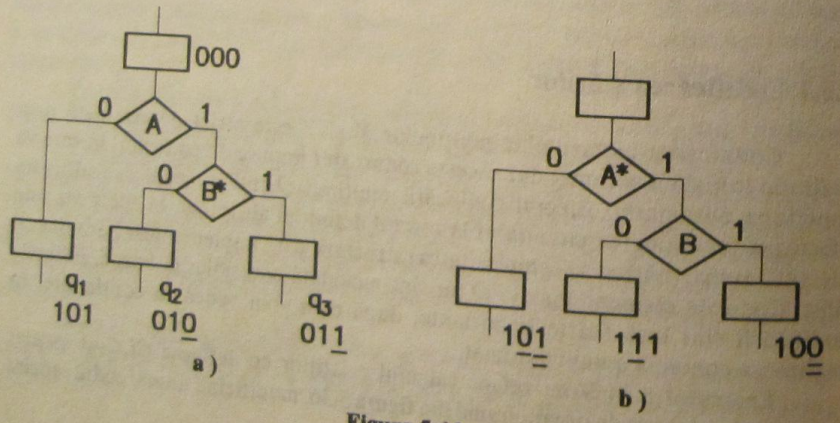



Figura 5.46



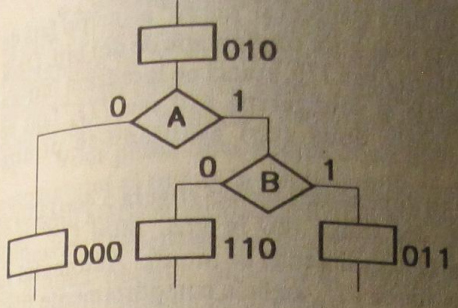



Figura 5.47

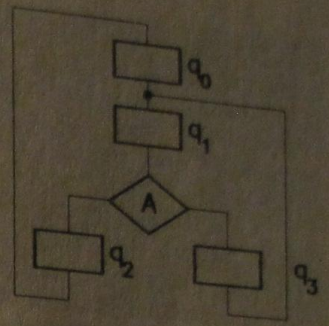



Figura 5.48

codurile stărilor distinse de $B$ sînt *01B*, numai cel mai puţin semnificativ bit depinzînd de intrarea $B$. În fragmentul de organigramă din figura 5.46*b* codificarea este realizată cu dependenţă redusă faţă de $A$. Pentru $B=0$ codurile stărilor alese prin $A$ sînt *1A1*, iar pentru $B=1$ sînt *10A'*. □

Putem realiza o codificare cu dependenţă redusă atît faţă de $A$ cît şi faţă de $B$, în exemplul anterior? Cititorul se poate singur convinge că nu.

• **Codificarea stărilor cu variaţie minimă** presupune ca stările succesive să fie astfel codificate încît să difere printr-un număr minim de biţi; dacă se poate numai printr-unul singur.

**Exemplul 5.16** Figura 5.47 exemplifică, printr-un fragment de organigramă, codificarea cu variaţie minimă. Starea *010* poate fi urmată de stări codificate prin *000*, *110* sau *011*, ce au proprietatea că diferă faţă de prima numai printr-un singur bit. □

Nu în orice situaţie o astfel de codificare este posibilă. Un singur exemplu este suficient pentru a ne convinge de acest lucru.

**Exemplul 5.17** Organigrama din figura 5.48 nu permite o codificare cu variaţie minimă pentru toate stările. Într-adevăr: dacă $q_1$ diferă de $q_0$ printr-un singur bit iar faţă de $q_2$ tot printr-un singur bit, atunci $q_0$ şi $q_2$ diferă cel puţin prin doi biţi. Deci pe o buclă de trei stări nu poate fi realizată codificarea cu variaţie minimă în cazul cel mai favorabil (al diferenţei de un singur bit). □

Cele două modalităţi de codificare sînt, evident, incompatibile. Ele pot fi folosite împreună la codificarea stărilor unui automat, dar numai în porţiuni distincte ale organigramei.

• **Automatele cu intrări asincrone** vor fi proiectate astfel încît codificarea stărilor să respecte principiul dependenţei reduse faţă de variaţia variabilelor de intrare ce nu comută sincron cu ceasul automatului. Să explicăm!

Vom lua în consideraţie fragmentul de organigramă din figura 5.47 cu $A=1$ şi $B$ o variabilă de intrare ce comută asincron faţă de ceas. Starea ce urmează stării *010* va fi, în aceste condiţii, *B'1B*. Bitul cel mai semnificativ al stării va fi calculat de un circuit logic iar cel mai puţin semnificativ de alt circuit, rezultînd din aceasta faptul că ieşirile $Q_2^+$ şi $Q_0^+$ nu vor comuta sincron una faţă de cealaltă. Mai mult, timpii de set-up ai bistabililor asociaţi celor doi biţi de stare nu pot fi identici. De aici rezultă că starea ce urmează stării *010* poate fi, în egală măsură, *010*, *011*, *110*

sau *111*, deoarece frontul activ al ceasului fiind complet şi "diferit" asincron faţă de comutarea ieşirilor $Q_2^+$ şi $Q_0^+$ poate surprinde orice configuraţie de doi biţi. Dacă în bistabilii de stare se înscrie configuraţia *011* sau *110*, atunci ne încadrăm într-un comportament firesc. Dar se poate înscrie şi configuraţia *010* sau *111*, fapt ce corespunde unei tranziţii eronate, aceste două stări neaflîndu-se printre succesoarele stării *010* cînd $A=1$.

Ce se întîmplă în cazul codificării cu dependenţă redusă (figura 5.46*a*) într-o situaţie identică. Automatul va comuta din starea *000* în *01B*. Bitul $Q_0^+$ este singurul care depinde, în acest caz, de variabila asincronă de intrare. Automatul va trece, în consecinţă, în starea *010* sau în starea *011*, ambele comportamente fiind perfect acceptabile. Indiferent cînd comută $B$, nu va exista o altă posibilitate de tranziţie.

Deci, dacă funcţia de tranziţie a stării depinde în fiecare moment numai de cel mult *o singură variabilă de intrare asincronă*, atunci este asigurată evoluţia corectă a automatului în spaţiul stărilor. Vom folosi, în consecinţă, codificarea cu dependenţă redusă pentru a evita efectele disfuncţionale ale intrărilor asincrone.

• **Sincronizarea variabilelor de intrare nu este cumva întotdeauna posibilă?** Vom elucida această problemă în continuare, dar pînă atunci cititorul poate să mediteze la un răspuns.

• **Hazardul la ieşirea automatelor imediate** este o altă problemă ce poate fi evitată printr-o codificare adecvată. Am arătat, anterior, că hazardul la ieşirea unui *CLC* este iminent atunci cînd la intrare comută mai mult decît o variabilă binară. De asemenea se pot crea premizele evitării lui atunci cînd la intrare comută numai un singur bit.

La ieşirea unui automat Moore imediat se pot produce tranziţii nedorite a căror evitare se poate realiza printr-o recodificare convenabilă a stărilor. Evident singura codificare convenabilă în acest caz ar fi una cu variaţie minimă realizată la limită, în sensul că orice tranziţie presupune comutarea a cel mult unui singur bit de stare. În acest caz se creează numai premizele eliminării hazardului. Vom exemplifica în paragraful 5.5.3.

Nu întotdeauna este necesară evitarea hazardului la ieşirea unui automat. Sînt cazuri în care aceste ieşiri sînt "interpretate" pur combinaţional, nefiind folosite pe intrări de ceas sau de *set, reset, load* ş.a. Dar, atunci cînd este necesar, putem evita hazardul prin acest procedeu al codificării cu variaţie minimă.

Putem evita hazardul la ieşirea unui automat Mealy imediat? Merită să gîndim puţin asupra acestui subiect.

• **Codificările combinate** pot fi uneori utile. În cazul în care intrările sînt sincrone iar problema hazardului la ieşire nu se pune (nu ne deranjează sau automatul este cu întîrziere şi hazardul nu poate apare) vom putea folosi libertatea oferită de posibilitatea diferitelor codificări pentru a optimiza *CLC*-ul asociat.

Într-o zonă a organigramei în care tranziţiile sînt preponderent condiţionate vom prefera o codificare cu dependenţă redusă, pentru ca în diagramele de tranziţie să apară în cît mai mică măsură variabilele de intrare. De asemenea, vom prefera codificarea cu variaţie minimă în cazul tranziţiilor necondiţionate deoarece diagramele de tranziţie astfel obţinute diferă minim de diagrama de referinţă. Se obţin astfel premizele unor funcţii de tranziţie simple.



• **Codurile nefolosite** pentru a denumi stările sînt asociate unor stări posibile în care automatul poate ajunge la punerea sub tensiune a circuitului sau printr-o funcţionare incorectă. Aceste coduri pot apare în registrul de stare, deci pot avea asociate stări ce nu vor putea fi evitate. În etapa de proiectare va trebui să ţinem cont de ele şi să definim o evoluţie oarecare în spaţiul acestor stări parazite.

Vom putea opta pentru două tipuri de evoluţii. O primă variantă permite automatului să revină în spaţiul stărilor utile. A doua variantă păstrează automatul în spaţiul stărilor parazite. Ambele variante trebuie luate în consideraţie în diverse contexte de funcţionare a automatului. Prima asigură o "resetare" automată iar cea de a doua permite punerea în evidenţă a unei disfuncţionalităţi prin blocarea automatului într-un ciclu eronat.

## 5.5.2 Limite fundamentale

Descrierea formală a automatelor ne permite să asociem unui automat orice evoluţie, în spaţiul stărilor şi al ieşirilor, sub influenţa intrărilor. Apar totuşi anumite restricţii în ceea ce priveşte cuplarea automatelor în sistemele din care fac parte. Unele limite se vor referi la cuplarea la intrare iar celelalte vor impune o restricţie referitoare la ieşire.

• **Cuplarea asincronă la intrare a mai multor variabile binare** pune o problemă insolubilă. Este vorba de *imposibilitatea realizării concrete a unui automat ale cărui tranziţii se realizează*, măcar dintr-o singură stare, *prin testarea a două sau mai multe variabile asincrone*.

Am arătat anterior că tranziţia dintr-o stare prin testarea unei variabile asincrone nu se poate face corect decît printr-o codificare cu dependenţă redusă. Putem realiza codificarea prin dependenţă redusă în cazul în care testăm două, sau mai multe, variabile asincrone? Să încercăm o astfel de codificare considerînd că atît $A$ cît şi $B$, din fragmentul de organigramă reprezentat în figura 5.46, sînt variabile asincron cuplate la intrarea automatului. Observăm că acest lucru este imposibil deoarece, pe de o parte stările $q_2$ şi $q_3$ diferă numai printr-un singur bit, $Q_0$, iar faţă de $q_1$, ar trebui să difere, la fel, tot printr-un singur bit, altul de fiecare dată. Acest lucru este imposibil deoarece atît pentru $q_2$ cît şi pentru $q_3$, $Q_2Q_1 = 01$, pentru a asigura dependenţa redusă faţă de $B$. Dependenţa redusă şi faţă de $A$ rezultă ca fiind imposibil de obţinut în acest caz. Dependenţa redusă faţă de $A$ implică o diferenţă de doi biţi pentru codurile stărilor $q_2$ şi $q_3$, fapt ce intră în contradicţie cu dependenţa redusă faţă de $B$ ce implica o diferenţă numai de un singur bit.

Deci nu putem testa coerent două variabile asincrone într-o singură stare.

Ar merita atunci să încercăm sincronizarea variabilelor asincrone. Pentru aceasta ar trebui introdus pe intrarea automatului un registru de doi biţi, comandat cu acelaşi ceas ca automatul. La intrarea acestui registru se pot conecta variabilele asincrone $A$ şi $B$. Rezultă o sincronizare a lor? Aparent da. Dar numai a fiecăreia în parte nu şi a celor patru configuraţii binare reprezentabile cu aceşti doi biţi. Să presupunem, prin absurd, că $A$ şi $B$ sînt sincrone între ele, dar asincrone faţă de

ceasul registrului. Dacă vor comuta în afara intervalului delimitat de $t_{SU}$ şi $t_H$ (figura 4.32) atunci configuraţia binară reprezentată de cei doi biţi se va înscrie corect în cei doi bistabili. Dar dacă $A$ şi $B$ vor comuta în acest interval împreună, atunci nu se ştie sigur dacă ambii bistabili vor putea sesiza această comutare. Dacă o sesizează ambii sau nici unul totul este în regulă. Dar dacă unul o sesizează iar celălalt nu, atunci în bistabili se va înscrie o configuraţie binară ce nu a fost niciodată prezentă la intrarea registrului.

Spre exemplu, la intrarea registrului, $A$ şi $B$ iau succesiv următoarele valori:

$$(00), (00), (01), (01), (10), (10),$$

la ieşire se vor obţine valorile

$$(00), (00), (00 \text{ sau } 01), (01), (00 \text{ sau } 01 \text{ sau } 10 \text{ sau } 11), (10).$$

Preluarea unei valori poate să fie întîrziată sau nu, cum este cazul tranziţiei de la *00* la *01*, cînd *se modifică un singur bit*. În cazul în care se modifică doi biţi în secvenţa de la ieşirea registrului pot apare configuraţii binare ce nu au existat la intrare. Astfel, tranziţia de la *01* la *10* poate fi preluată corect imediat sau corect întîrziat, dar mai poate fi preluată şi eronat cînd numai tranziţia bitului $B$ este sesizată (în intervalul $t_{SU}$ - $t_H$) sau cînd numai tranziţia bitului $A$ este preluată. În aceste din urmă cazuri în registru se înscrie *00*, respectiv, *11*. Automatul va recepţiona astfel o configuraţie binară ce nu se regăseşte în şirul celor asincrone de la intrarea registrului. *Prin sincronizarea unei configuraţii binare asincrone de cel puţin doi biţi se pot introduce în şirul sincronizat configuraţii binare ce nu există în secvenţa asincronă.*

Deci, sincronizarea intrărilor asincrone cu un registru nu este o soluţie. Reţinem limitarea fundamentală ce nu ne permite cuplarea unor variabile asincrone la intrarea unui automat ce posedă tranziţii condiţionate de mai mult decît o variabilă asincronă. Acest fapt nu împiedică cuplarea la intrarea unui automat finit a mai mult de o singură variabilă asincronă, dar trebuie respectată restricţia ca în nici o stare printre variabilele testate să nu fie mai mult decît una asincronă.

• **Un automat Mealy imediat fără hazard pe ieşire este o imposibilitate.** Afirmaţia este imediat justificabilă prin faptul că la fiecare tranziţie poate să comute, pentru a asigura o evoluţie în spaţiul stărilor, cel puţin un bit de stare la care oricînd se adaugă variaţia unui bit de intrare. Deci nu putem evita situaţia în care *CLC*-ul ce calculează funcţia $\delta$ suportă pe intrare variaţia a doi biţi, caz în care hazardul este inevitabil.

Deci nu ne putem propune proiectarea unui Mealy imediat fără hazard pe ieşire.

• **Automatul Mealy cu întîrziere care admite variabile de intrare asincrone** nu poate fi realizat din motive similare celor ce nu permit două variabile asincrone simultan testate. În cazul codurilor folosite pentru mulţimea $Y$ nu se pot impune restricţii de codificare; aceste coduri sînt date prin definirea automatului şi sînt impuse de aplicaţie, de modul de cuplare în sistem. Oricînd vor putea apare tranziţii ale ieşirii la care cel puţin un bit depinde de o variabilă asincronă. Deci în registrul automatului se vor înscrie doi biţi, unul de stare şi unul de ieşire, care



depind de o variabilă asincronă de intrare. În acest caz putem ajunge în situaţia în care în registru se află o configuraţie de ieşire ce corespunde unei alte stări decît cea desemnată prin definiţie. Dacă doi, sau mai mulţi, biţi de ieşire depind de o variabilă asincronă, atunci se poate întîmpla ca, chiar configuraţia de ieşire să fie eronat înscrisă în registrul de ieşire.

• **Concluzionînd** putem spune că *un automat Mealy cu intrări asincrone nu poate fi acceptat, în cazul general, ca o realizare practică.* Cel cu întîrziere nu poate funcţiona conform definiţiei. Cel imediat generează pe ieşire semnale încărcate cu *hazard asincron*. În mod evident, hazardul asincron face ca un semnal logic să fie inutilizabil. Atunci cînd hazardul apare într-un interval de timp previzibil el poate fi evitat printr-o bună preluare a semnalului, altfel, semnalul avînd un comportament absolut imprevizibil devine inutilizabil.

Automatele Mealy au sens numai cuplate sincron pe intrare, fiind utile în aplicaţiile ce impun o reacţie promptă (în ciclul curent) şi rapidă (direct de la ieşirea unui registru).

Automatele Moore admit variabile de intrare asincrone, decuplarea dintre intrare şi ieşire fiind mult mai mare decît în cazul automatelor de tip Mealy. Acest avantaj este "plătit" prin creşterea dimensiunii spaţiului stărilor.

• **Automatul Moore imediat cu intrări asincrone şi ieşiri fără hazard** reprezintă, în cazul general, o imposibilitate. Acest lucru este evident deoarece intrările asincrone impun o codificare a stărilor cu dependenţă redusă iar eliminarea hazardului pe ieşiri cere o codificare cu variaţie minimă. Îndeplinirea ambelor restricţii este imposibilă şi în consecinţă, un astfel de automat este imposibil de proiectat, cu excepţia unor cazuri limită, particulare, ce vor fi exemplificate în paragraful următor.

În *general* problema unui automat cu intrări asincrone şi ieşiri fără hazard îşi găseşte soluţia într-un automat de tip Moore cu întîrziere. Preţul plătit, ce uneori poate fi acceptat, este al unei reacţii întîrziate, cu două cicluri de ceas, faţă de variaţia intrării.

### 5.5.3 Exemple de proiectare

Aplicarea considerentelor teoretice legate de proiectarea automatelor finite se cere amplu exemplificată deoarece o serie de tehnici concrete sînt mai greu de teoretizat abstract.

Un prim exemplu va încerca evidenţierea efectului pe care-l au cele două tipuri de codificări asupra complexităţii structurii interne a automatului, deci asupra semiautomatului.

**Exemplul 5.18** În figura 5.49 este reprezentată organigrama unui semiautomat cu patru stări. Care din multiplele posibilităţi de codificare a stărilor este cea mai eficientă pentru structura internă a semiautomatului? Neimpunîndu-se restricţii asupra modului de conectare putem realiza orice fel de codificare. Sperăm că următoarele principii de codificare să conducă la minimizarea structurii *CLC*-ului asociat buclei:

- în zonele în care tranziţia este necondiţionată să se adopte o codificare cu variaţie minimă;

- în zonele în care tranziţia este condiţionată să se aplice principiul dependenţei reduse (astfel în diagramele de tranziţie vor apare într-un număr minim de poziţii variabilele de intrare).

Aplicînd acest mod de codificare rezultă codurile din parantezele ronde pe figura 5.49. În figura 5.50 diagrama notată cu $(Q_1^+Q_0^+)$ reprezintă funcţia de tranziţie pentru această primă codificare. Rezultă

$$Q_1^+ = Q_0$$
$$Q_0^+ = Q'_1 + Q_0 A'.$$

Dacă se aplică o codificare ce ignoră complet principiile anterior enunţate, atunci rezultă codificarea notată în paranteze drepte şi diagrama $[Q_1^+Q_0^+]$ în figurile 5.49 şi 5.50, din care se extrag funcţiile logice de mai jos

$$Q_1^+ = Q_1 Q_0 + Q'_1 Q'_0 + Q_1 A$$
$$Q_0^+ = Q'_1 Q'_0 + Q'_0 A.$$

Valabilitatea regulilor anterior enunţate este evidentă. Ele sînt numai nişte *reguli* ce nu pot fi întotdeauna riguros şi complet aplicate. Ne putem însă orienta după ele pentru a găsi soluţii mai simple, dacă nu se va putea ca întotdeauna să fie minimale. □

Aplicarea regulii de codificare cu variaţie minimă nu este întotdeauna simplă, necesitînd, atunci cînd o configuraţie particulară a organigramei o impune, reformulări importante ale enunţului problemei.

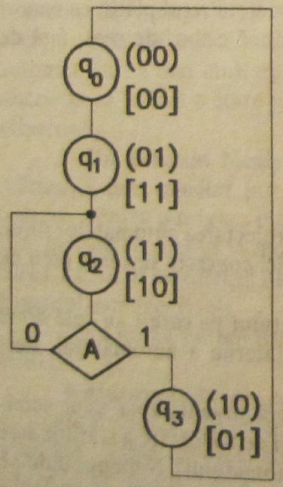



Figura 5.49

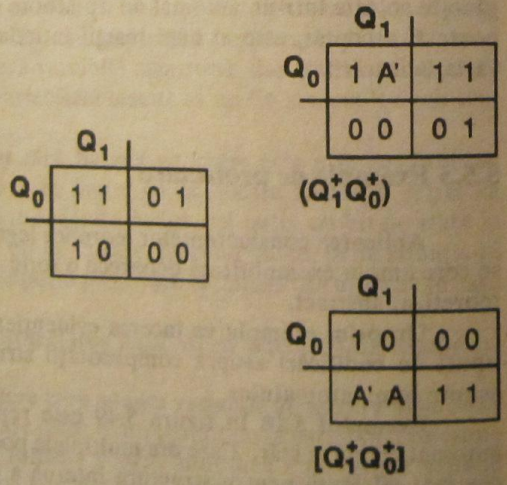



Figura 5.50



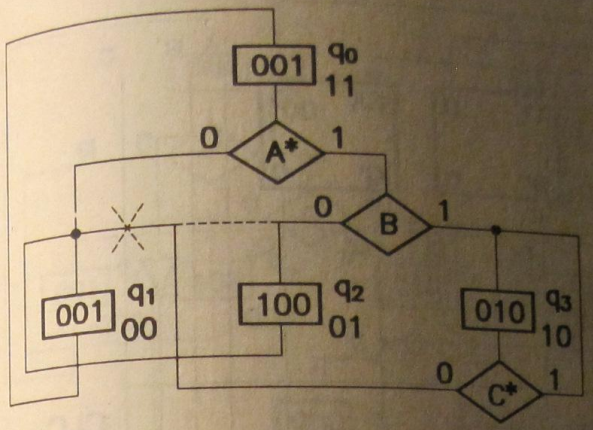



Figura 5.51

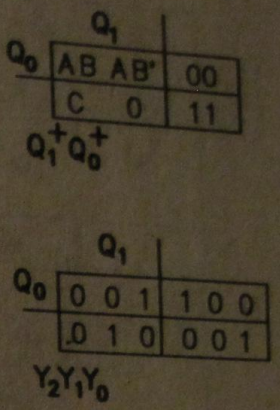



Figura 5.52

**Exemplul 5.19** Organigrama din figura 5.51 descrie funcţionarea unui automat Moore cu variabilele de intrare $A$ şi $C$ cuplate asincron. Variabila $B$ este sincronă cu ceasul ce comandă automatul. Vom fi obligaţi să codificăm stările conform principiului dependenţei reduse faţă de variabilele asincrone $A$ şi $C$. (Ignorăm, în acest exemplu, liniile punctate din figura 5.51.) Rezultă codificarea binară notată pe organigramă. Astfel, starea $q_0$ este urmată de $0A$ dacă $B=0$ şi de $A0$ dacă $B=1$. Starea $q_3$ este urmată de $C0$. Rezultă diagramele de tranziţie reprezentate în figura 5.52, din care extragem următoarele funcţii logice:

$$Q_1^+ = Q'_1 Q'_0 + Q'_0 C + Q_1 Q_0 AB,$$
$$Q_0^+ = Q'_1 Q'_0 + Q_1 Q_0 AB',$$
$$Y_2 = Q'_1 Q_0,$$
$$Y_1 = Q_1 Q'_0,$$
$$Y_0 = Q_1 Q_0 + Q'_1 Q'_0$$

care definesc $CLC$-ul asociat automatului.

Pentru a implementa automatul în varianta ce asigură viteza de lucru maximă (cu adîncime minimă pentru $CLC$), vom rescrie ecuaţiile după cum urmează:

$$Q_1^+ = ((Q'_1 Q'_0)'(Q'_0 C)'(Q_1 Q_0 AB)')',$$
$$Q_0^+ = ((Q'_1 Q'_0)'(Q_1 Q_0 AB')')',$$
$$Y_2 = ((Q'_1 Q_0)')',$$
$$Y_1 = ((Q_1 Q'_0)')',$$
$$Y_0 = ((Q_1 Q_0)'(Q'_1 Q'_0)')'.$$

Ecuaţiile pentru $Y_2$ şi $Y_1$ s-au complicat, aparent, inutil. Le preferăm sub această formă pentru că asigură o realizare concretă cu aceleaşi tipuri de porţi logice, fără a se introduce o întîrziere globală suplimentară. Schema ce rezultă este

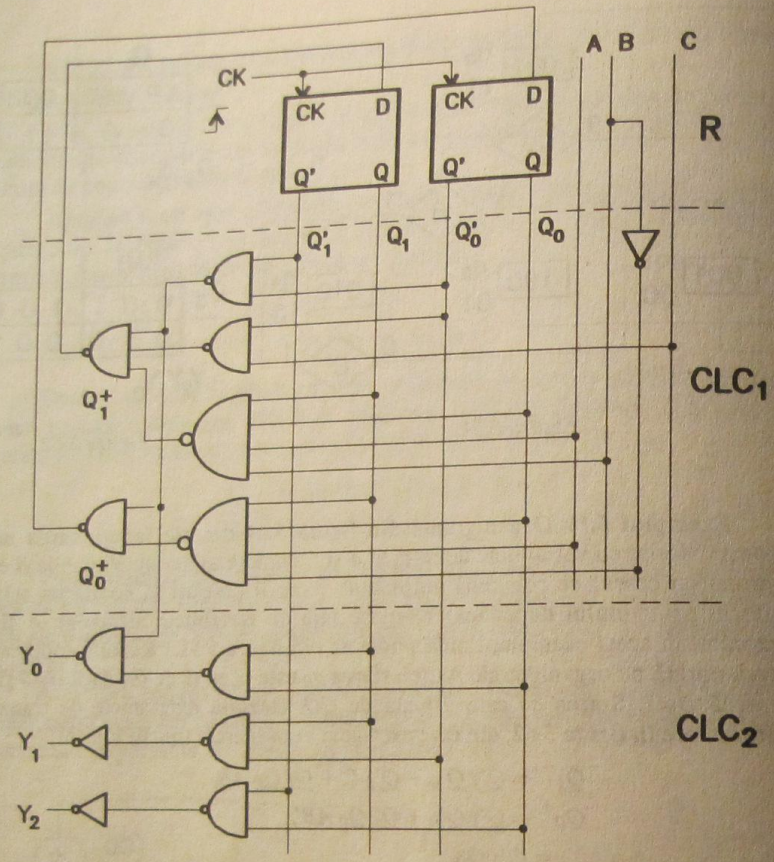



Figura 5.53

reprezentată în figura 5.53, unde au fost demarcate prin linii punctate zonele ce corespund registrului $R$, şi circuitelor combinaţionale $CLC_1$ şi $CLC_2$ conform reprezentării automatului Moore imediat din figura 5.31c. □

Există configuraţii de organigramă ce nu permit realizarea unei codificări cu dependentă redusă. Pentru aceste situaţii, uneori, putem găsi o soluţie prin *adăugarea unei stări suplimentare*.

**Exemplul 5.20** Vom considera organigrama ce rezultă luînd în consideraţie liniile punctate din figura 5.51. Rezultă că starea ce urmează stării $q_3$, pentru $C=0$, va fi $q_2$ şi nu $q_1$. O simplă analiză ne arată că în acest caz nu se poate asigura o codificare cu dependenţă redusă faţă de $A$ şi $C$ simultan. Stările $q_2$ şi $q_3$ trebuie să difere între ele prin doi biţi pentru a se asigura dependenţa redusă faţă de $A$, atît pentru $B=0$ cît şi pentru $B=1$. Pentru $B=0$ succesorul lui $q_0$ este $0A$ iar pentru $B=1$ este $A0$. Deoarece $q_1=00$ rezultă că putem alege codurile lui $q_2$ şi $q_3$ din mulţimea $\{01, 10\}$. În consecinţă, nu mai există soluţie pentru a fi asigurată dependenţa



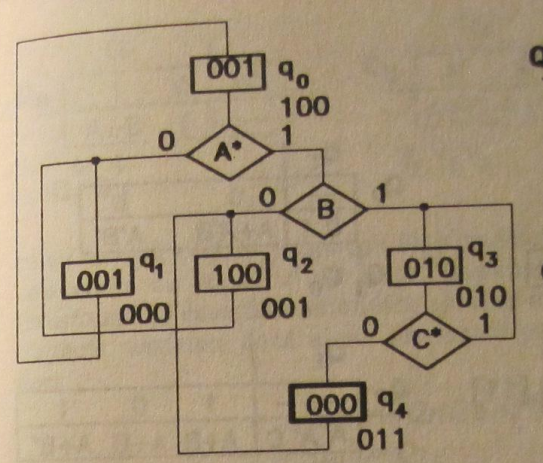



Figura 5.54

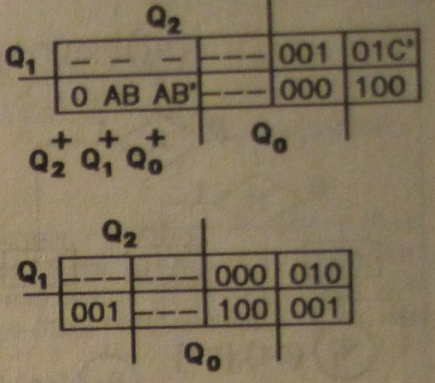



Figura 5.55

redusă faţă de $C$, pentru că tot $q_2$ şi $q_3$ sînt şi stările ce-i succed lui $q_3$ distinse prin variabila asincronă $C$.

Vom soluţiona problema introducînd o stare suplimentară, $q_4$, conform reprezentării din figura 5.54. În această stare ieşirea va fi *000* (dacă aplicaţia permite acest lucru; dacă nu, se va repeta ieşirea din $q_3$, *010*, tot dacă aplicaţia permite; iar dacă nici acest lucru nu este posibil ne aflăm iarăşi în faţa unei limite fundamentale). În acest moment codificarea este posibilă deoarece succesorii lui $q_3$ sînt $q_3$ şi $q_4$, codificaţi prin *01C'*. Celelalte codificări (pentru tranziţiile necondiţionate) sînt realizate cu variaţie minimă. Rezultă diagramele de tranziţie din figura 5.55 şi ecuaţiile

$$Q_2^+ = Q'_2 Q'_1 Q'_0,$$
$$Q_1^+ = Q_1 Q'_0 + Q_2 AB,$$
$$Q_0^+ = Q_1 Q_0 + Q_1 C' + Q_2 AB',$$
$$Y_2 = Q'_1 Q_0,$$
$$Y_1 = Q_1 Q'_0,$$
$$Y_0 = Q_2 + Q'_1 Q'_0,$$

ce descriu circuitele logice asociate buclei ($Q_2^+$, $Q_1^+$, $Q_0^+$) şi ieşirii ($Y_2$, $Y_1$, $Y_0$).

În starea $q_4$ automatul generează comanda *noop* (no operation) codificată prin *000*. Dacă în această stare s-ar fi menţinut comanda anterioară (bitul $Y_1$ ar fi avut un efect ce nu depindea şi de durata sa) atunci s-ar fi modificat numai funcţia $Y_1$, simplificîndu-se la $Y_1 = Q_1$. □

După cum am arătat în paragraful anterior automatele de tip Mealy nu au sens decît cu variabile sincrone pe intrare. Problema hazardului nu este, de asemenea, semnificativă deoarece pentru cele imediate nu poate fi nici eliminat, nici tolerat cu variabile asincrone, iar la cele cu întîrziere problema nu se pune.

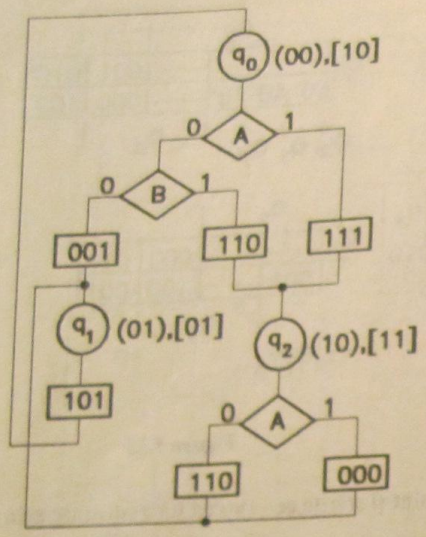



Figura 5.56

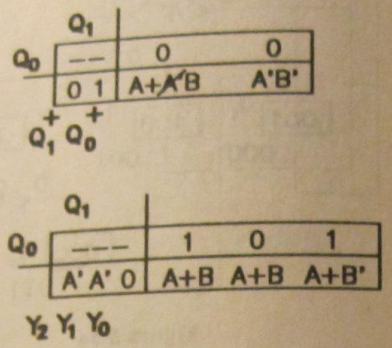



Figura 5.57

**Exemplul 5.21** Vom implementa automatul Mealy descris de organigrama din figura 5.56. Conform unei prime codificări, cea notată în figură între paranteze ronde, se obţin diagramele de tranziţie din figura 5.57, din care extragem următoarele ecuaţii:

$$Q_1^+ = Q'_1 Q'_0 (A+B),$$
$$Q_0^+ = Q_1 + Q'_0 A'B',$$
$$Y_2 = Q_0 + Q_1 A' + Q'_1 (A+B),$$
$$Y_1 = Q_1 A' + Q'_1 Q'_0 (A+B),$$
$$Y_0 = Q_0 + Q'_1 (A+B').$$

Încercăm o codificare mai abilă pentru a minimiza complexitatea *CLC*-ului, deoarece alte restricţii nu se impun (legate de asincronism sau hazard). Reuşim o codificare cu dependenţă redusă la tranziţia din $q_0$ şi una cu variaţie minimă la comutarea din $q_2$ (conform notaţiilor din parantezele drepte). Rezultă diagramele de tranziţie din figura 5.58 şi ecuaţiile următoare:

$$Q_1^+ = Q'_1 + Q'_0 (A+B),$$
$$Q_0^+ = Q_1,$$
$$Y_2 = Q'_1 + Q_0 A' + Q'_0 (A+B),$$
$$Y_1 = Q_1 Q_0 A' + Q'_0 (A+B),$$
$$Y_0 = Q'_1 + Q'_0 (A+B'),$$



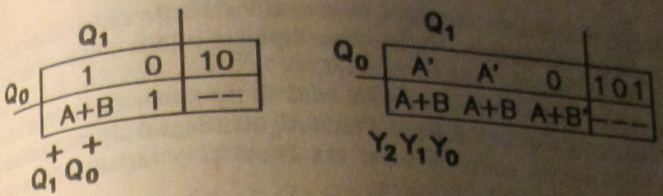



Figura 5.58

ce au asociat, evident, un circuit logic mai simplu, reprezentat în figura 5.59, în conexiune cu registrul de stare. Implementarea automatului presupune transformarea ecuaţiilor anterioare aplicînd teorema lui De Morgan, astfel ca să obţinem maximum două nivele inversoare pe bucla combinaţională. Vom scrie deci,

$$Q_1^+ = (Q_1(Q'_0A)'(Q'_0B)')',$$
$$Q_0^+ = Q_1,$$
$$Y_2 = (Q_1(Q_0A')'(Q'_0A)'(Q'_0B)')',$$
$$Y_1 = ((Q_1Q_0A')'(Q'_0A)'(Q'_0B)')',$$
$$Y_0 = (Q_1(Q'_0A)'(Q'_0B')')'.$$

Se poate observa că în a doua variantă, printr-o "mai bună" codificare, se obţine, pentru semiautomat, un *CLC* de 1,67 ori mai puţin complex decît în prima variantă. Atunci cînd codificarea nu este restricţionată de criterii ce ţin de cuplarea

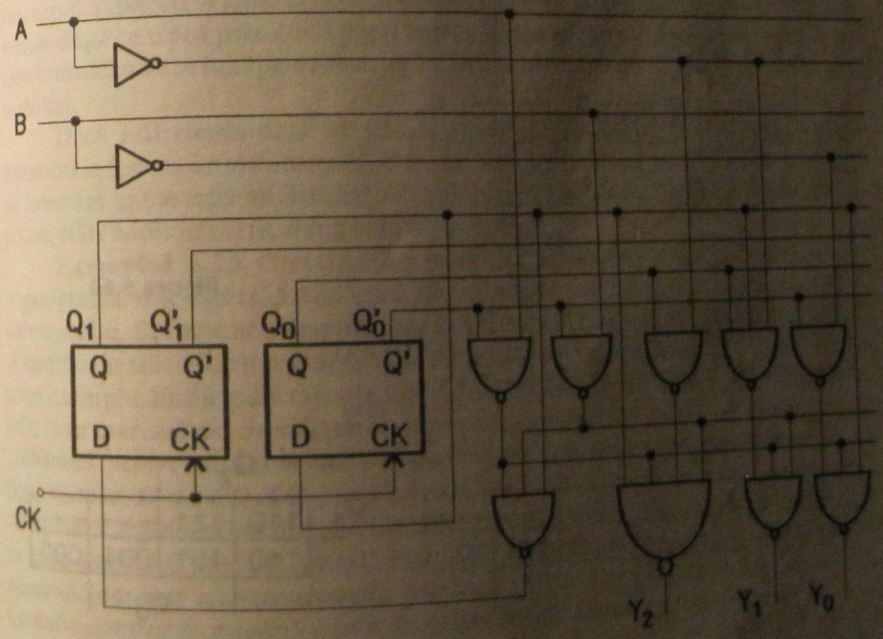



Figura 5.59

automatului în sistem, vom folosi libertatea de a codifica stările astfel încît să minimizăm structura semiautomatului (Nu este sigur faptul că soluţia obţinută este minimală. Cititorul poate încerca şi alte variante.) □

Automatul anterior putea fi realizat şi ca unul de tip Mealy cu întîrziere. Pentru aceasta era suficientă cuplarea la ieşire a unui registru de trei biţi comandat cu acelaşi ceas ca registrul de stare. În acest caz există şi o variantă Moore echivalentă. O vom studia în exemplul următor.

**Exemplul 5.22** Organigrama din figura 5.56 poate fi translatată într-una echivalentă pentru un automat Moore. Acest lucru este realizat în figura 5.60. Evoluţia ieşirii celor două automate este identică sub acţiunea unor intrări identice. Am început desenarea acestei organigrame din starea $q_1$ a automatului Mealy, starea în care ieşirea este *101*. Acest cod este generat de automatul Moore în starea $q_0$. Ieşirea următoare este determinată de valorile luate de simbolurile binare $A$ şi $B$. Deci stării $q_0$ îi urmează *trei* stări în varianta Moore: $q_1$, $q_2$ şi $q_3$. Astfel, din aproape în aproape, se poate genera organigrama asociată variantei Moore.

Codificarea celor şase stări obţinute se va face în aşa fel încît *diferenţa între* $Q_2Q_1Q_0$ *şi* $Y_2Y_1Y_0$ *să fie minimală.* (Uneori ne putem permite ca $Q \equiv Y$ !). Observăm că ieşirile sînt diferite în cele şase stări cu excepţia stărilor $q_2$ şi $q_4$ în care

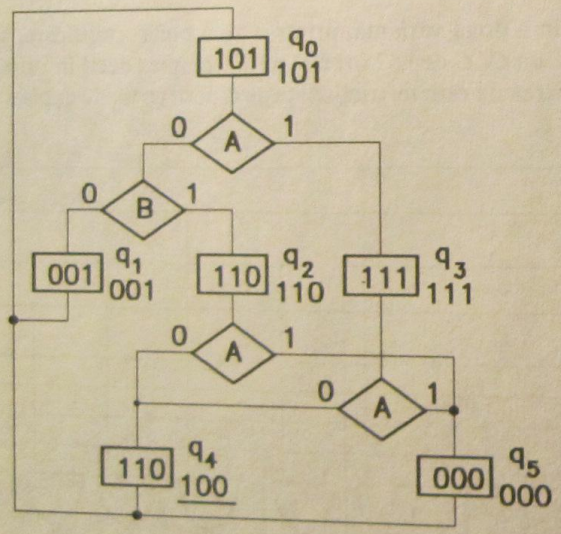



Figura 5.60

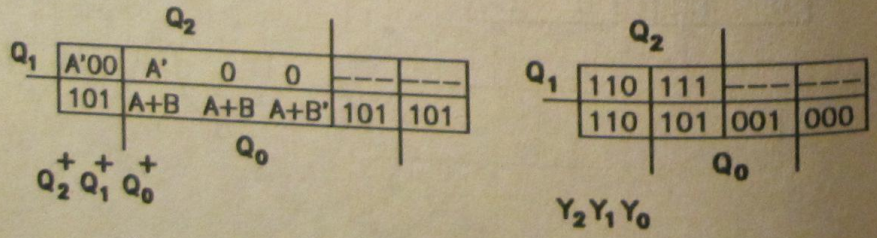



Figura 5.61



ieşirea este aceeaşi, *110*. Vom acorda stării $q_2$ codul *110* iar pentru $q_4$ vom alege un cod ce diferă minim de *110* şi nu este folosit pentru desemnarea nici unei alte ieşiri sau stări. Alegem *100*. Pentru celelalte stări folosim drept coduri configuraţiile binare de ieşire asociate. Rezultă codificarea notată pe figura 5.60, conform căreia pot fi generate diagramele de tranziţie asociate automatului (figura 5.61), din care extragem ecuaţiile de mai jos:

$$Q_2^+ = Q'_1 Q'_0 + Q'_2 + Q_1 A' + Q'_1 (A + B),$$
$$Q_1^+ = Q_2 Q'_1 Q_0 (A+B),$$
$$Q_0^+ = Q'_2 + Q'_1 Q'_0 + Q'_1 (A+B'),$$
$$Y_2 = Q_2,$$
$$Y_1 = Q_1 + Q_2 Q'_0,$$
$$Y_0 = Q_0.$$

Remarcăm simplitatea ecuaţiilor asociate ieşirii. Diferenţa de cod minimă de un bit, între $q_2$ şi $q_4$ a făcut ca numai un bit de ieşire ($Y_1$) să fie calculat printr-un circuit logic; ceilalţi doi corespund biţilor de stare. □

Circuitul rezultat este totuşi de *1,46* ori mai complex decît cel oferit de soluţia de tip Mealy. Nu trebuie să uităm că la automatul de tip Mealy cu întîrziere există un registru de *5* biţi (doi de stare şi trei de ieşire), pe cînd la automatul Moore imediat apar numai trei bistabili, cei ai registrului de stare. Pe de altă parte automatul Moore imediat prezintă riscul hazardului pe ieşire, risc pe care varianta Mealy nu-l presupune. Automatul Mealy mai prezintă avantajul că ieşirea comută mai rapid, raportat la frontul activ al ceasului, pentru că la Moore, $Y_1$ se stabilizează numai după ce trece prin două porţi logice. Alegerea între cele două soluţii o vom face întotdeauna în funcţie de restricţiile impuse de sistemul în care automatul este înglobat.

Dacă întîrzierea dată de $CLC_2$ (figura 5.31*c*) nu poate fi evitată, totuşi, hazardul la ieşirea unui automat Moore imediat poate fi eliminat, chiar dacă uneori ne asumăm unele mici modificări ale definiţiei iniţiale (introducînd, atunci cînd se poate, stări suplimentare, după cum vom vedea).

**Exemplul 5.23** Considerăm automatul Moore imediat descris de organigrama din figura 5.62. Variabila de intrare $A$ variază sincron cu ceasul automatului. Se impune o implementare fără hazard la ieşire. Sîntem, deci, obligaţi să codificăm stările automatului aplicînd principiul variaţiei minime. (Ignorăm, în acest exemplu, liniile punctate din figura 5.62) Codificarea propusă pentru cele şase stări este marcată pe organigramă şi se poate verifica faptul că orice tranziţie se realizează prin comutarea unui singur bit (în general se pune problema ca să comute cel mult un bit). Diagramele de tranziţie sînt prezentate în figura 5.63, unde cele pentru ieşiri au fost distinct realizate pentru a putea fi uşor analizate situaţiile în care hazardul trebuie evitat. Prin codificarea realizată am înlăturat numai *hazardul datorat asincronismului intrărilor*. Urmează preocuparea legată de *hazardul datorat propagărilor* prin circuitele ce calculează ieşirea.

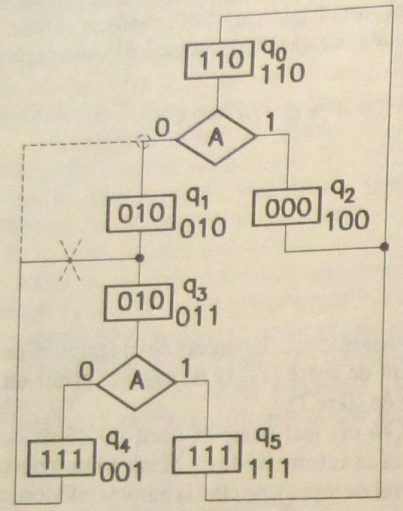



Figura 5.62

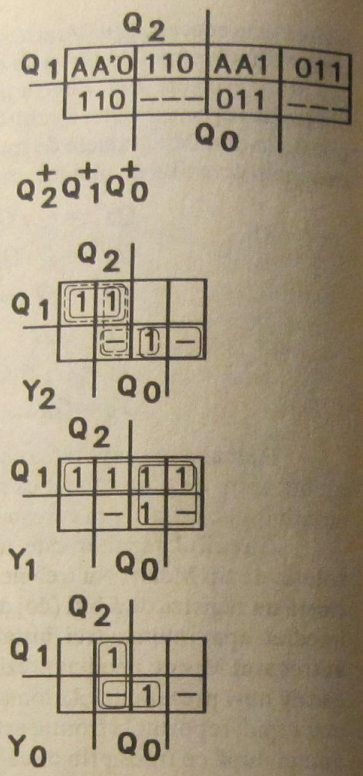



Figura 5.63

Ne vom concentra numai asupra ecuaţiilor ce descriu circuitele de ieşire. Funcţia pentru $Y_2$ poate fi extrasă în mai multe feluri din diagramă. Considerăm că singura acceptabilă este următoarea:

$$Y_2 = Q_1Q_2 + Q'_1Q'_2 .$$

O variantă posibilă ar fi

$$Y_2 = Q_1Q_2 + Q'_1Q_0 ,$$

dar aceasta, la comutarea bitului de stare $Q_1$, cînd $Q_2=1$ şi $Q_0=1$, are asociat un circuit care generează o tranziţie parazită. Altfel spus această formă nu elimină hazardul pe ieşire. Ea poate fi corectată luînd în consideraţie o suprafaţă suplimentară, dar atunci rezultă ecuaţia

$$Y_2 = Q_1Q_2 + Q'_1Q_0 + Q_2Q_0$$

ce este mai complexă decît prima formă pe care am considerat-o valabilă.

Pentru $Y_1$ rezultă, fără comentarii,



$$Y_1 = Q'_2 + Q_1 .$$

Ultimul bit de ieşire, $Y_0$, poate fi şi el extras în două maniere, după cum urmează

$$Y_0 = Q_2Q_0 + Q'_1Q_0 ,$$
$$Y_0 = Q_2Q_0 + Q'_2Q'_1 .$$

Cea de a doua formă este generatoare de hazard la comutarea variabilei $Q_2$ cînd $Q_1=0$ iar $Q_0=1$. Deci reţinem numai prima formă. □

Codificarea cu variaţie minimă poate fi şi ea stingherită de configuraţii particulare ale organigramelor, tot aşa cum şi codificarea cu dependenţă redusă a fost imposibilă într-o configuraţie particulară (Exemplul 5.20). Soluţia ce se dă acestor situaţii particulare este aceeaşi: *introducerea unei stări suplimentare*.

**Exemplul 5.24** De această dată vom considera varianta de organigramă realizată cu linie punctată în figura 5.62. Implementarea în condiţiile din exemplul precedent se loveşte de imposibilitatea codificării cu variaţie minimă pe bucla formată de stările $q_1$, $q_3$ şi $q_4$. Într-adevăr, $q_3$ ar trebui să se distingă printr-un bit de $q_4$ şi prin altul de $q_1$, deci $q_1$ se va distinge de $q_4$ prin doi biţi.

Soluţia ce se impune este introducerea unei stări suplimentare pe această buclă. În această stare se va genera în exterior *000*. (O variantă posibilă ar fi generarea ieşirii din starea anterioară, *111*.) Dacă această soluţie este acceptabilă în contextul în care funcţionează automatul, atunci mergem mai departe. Dacă nu, problema este insolubilă.

Considerînd că introducerea stării suplimentare reprezintă o perturbare acceptabilă, redesenăm organigrama ca în figura 5.64, introducînd starea $q_6$ cu

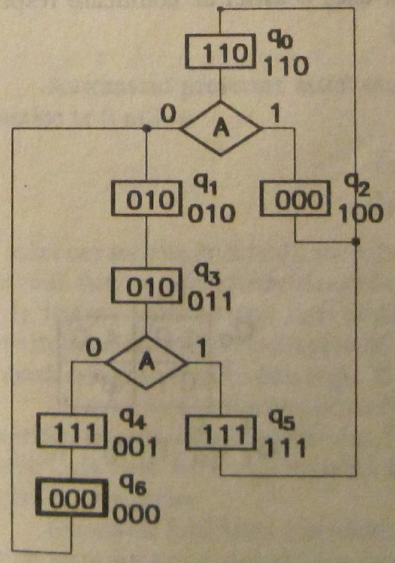



Figura 5.64

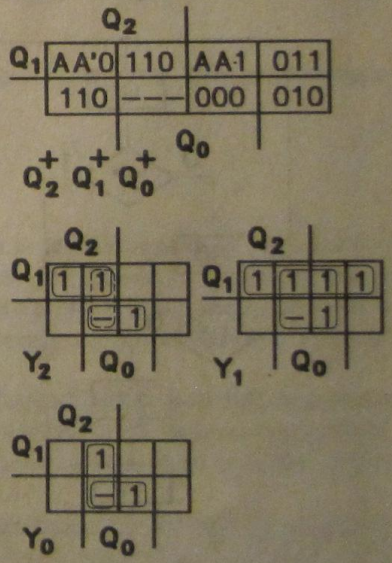



Figura 5.65

ieşirea *000*. Codificarea lui $q_6$ o facem prin *000*, asigurînd astfel variaţia minimă pentru orice tranziţie a automatului. Obţinem diagramele de tranziţie din figura 5.65, din care extragem funcţiile de ieşire, după cum urmează:

$$Y_2 = Q_2 Q_1 + Q'_1 Q_0 + Q_2 Q_0,$$

cu ultimul termen adăugat pentru evitarea hazardului,

$$Y_1 = Q_1 + Q_0,$$
$$Y_0 = Q_2 Q_0 + Q'_1 Q_0.$$

În cazul ultimelor două funcţii nu există riscul hazardului. □

Aşa cum există limite fundamentale ce împiedică realizarea şi utilizarea practică a unor tipuri de automate, există şi cazuri de excepţie, în care limitele fundamentale pot fi eludate. Am spus că, *în general*, nu putem construi un automat Moore imediat cu variabile asincrone pe intrare şi cu ieşiri neafectate de hazard. Există totuşi probleme particulare de acest tip care-şi găsesc soluţia.

**Exemplul 5.25** Ne propunem să construim automatul Moore imediat descris prin organigrama din figura 5.66, impunînd absenţa hazardului la ieşirile $Y_1$ şi $Y_0$ în situaţia în care variabilele de intrare $A$ şi $B$ sînt asincrone.

Aplicînd cele anterior expuse ar trebui să codificăm stările acestui automat aplicînd simultan principiul dependenţei reduse (pentru comutarea corectă sub comanda intrărilor asincrone) şi principiul variaţiei minime (pentru a crea premizele eliminării hazardului de la ieşire). Acest lucru este imposibil. Există totuşi, în cazul acestei probleme, o soluţie!

Dacă identificăm mulţimea stărilor cu cea a ieşirilor, $Q \equiv Y$, atunci vom putea "lua" ieşirile direct din registrul de stare şi nu se mai pune problema hazardului pentru că $CLC_2$ dispare. Rămîne să vedem dacă o astfel de codificare respectă

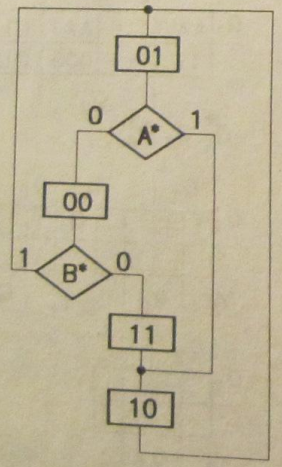



Figura 5.66

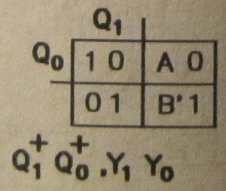



Figura 5.67



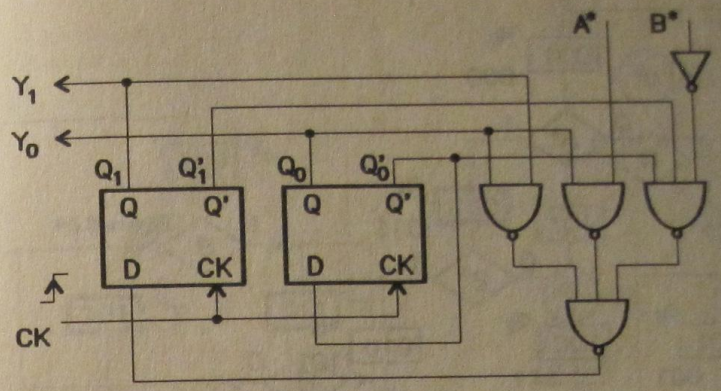



Figura 5.68

principiul dependenţei minime. Vom observa (ca din întîmplare!) că într-adevăr avem, în acest exemplu, codificarea dorită.

Diagrama unică de tranziţie, pentru stare şi ieşire, este reprezentată în figura 5.67. Din ea rezultă:

$$Q_1^+ = Q_1 Q_0 + Q_0 A + Q'_1 Q'_0 B,$$
$$Q_0^+ = Q'_0,$$

în condiţii în care

$$Y_1 = Q_1,$$
$$Y_0 = Q_0.$$

Automatul proiectat astfel este prezentat în figura 5.68. Din considerente practice ar fi util ca

$$Y_1 = (Q'_1)',$$
$$Y_0 = (Q'_0)'.$$

În acest caz ieşirile ar fi realizate prin inversoare cuplate la ieşirile $Q'$ ale celor doi bistabili. Sensul acestei soluţii este dat de faptul că o conexiune, ce riscă să fie lungă, de la ieşirea unui bistabil este periculoasă pentru buna funcţionare electrică a circuitului. Reflexiile ce pot apare pe un traseu lung pot comuta bistabilul "trăgînd" la masă o ieşire aflată în unu logic. □

Există şi alte limite fundamentale ce pot fi evitate prin uşoare reformulări ale definiţiei iniţiale date automatului. În această situaţie se află automatele definite astfel încît este necesară testarea într-o singură stare, a două sau mai multe variabile asincrone.

**Exemplul 5.26** Vom considera că organigrama dată în figura 5.51 (neglijînd liniile punctate) este definită cu variabilele $A$, $B$ şi $C$ asincrone. În acest caz din starea $q_0$ tranziţia este realizată condiţionată de două variabile asincrone, $A$ şi $B$.

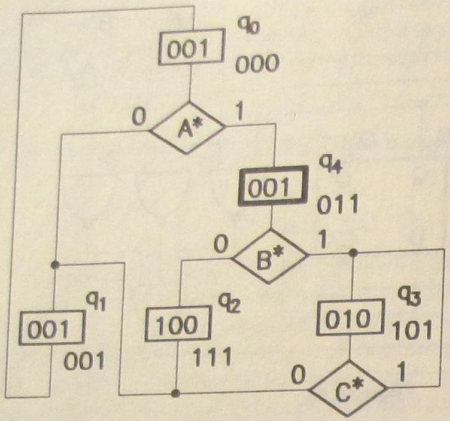



**Figura 5.69**

Sub această formă, neexistînd soluţie, vom modifica organigrama introducînd o stare suplimentară în aşa fel încît noul automat definit să funcţioneze prin testarea a cel mult unei singure variabile asincrone în fiecare stare. Vom introduce o stare suplimentară "înainte" de testarea variabilei $B$, conform reprezentării din figura 5.69. Starea $q_4$, suplimentar introdusă, permite ca variabilele asincrone $A$ şi $B$ să fie testate în stări distincte. Ieşirea în această stare va fi aceeaşi ca în $q_0$, în exteriorul automatului fiind sesizată o tranziţie "mai lentă" din $q_0$ în $q_2$ sau $q_3$. Dacă, acest lucru nu prejudiciază asupra funcţionării ansamblului în care automatul este introdus, atunci o astfel de soluţie poate fi acceptată. Se poate propune şi ca în $q_4$ ieşirea să fie "no operation", adică *000*, dacă o astfel de variantă are sens în context. □

Exemplele 5.21 şi 5.22 ne-au arătat că, cel puţin în anumite cazuri, un automat Mealy poate oferi o soluţie mai avantajoasă decît unul de tip Moore. Va fi, deci, interesant să reuşim, uneori măcar, să convertim un automat de tip Moore într-unul de tip Mealy. (Invers am văzut că este relativ simplu.) Nu întotdeauna printr-o astfel de conversie se obţine un automat Mealy cu un număr mai mic de stări.

**Exemplul 5.27** Pornind de la organigrama din figura 5.70 va trebui să construim organigrama automatului Mealy echivalent. Observăm că în stările $q_1$ şi $q_2$ se converge prin trei testări identice ale variabilei $B$. Dacă $B=0$ automatul comută în starea $q_1$ din $q_0$, $q_2$ şi $q_5$, iar dacă $B=1$ automatul comută în $q_2$ din aceleaşi stări.

Plecînd de la această observaţie vom începe desenarea organigramei cerute marcînd starea $q_0$ în care se testează $B$ şi se generează în exterior *101* şi *011* (figura 5.71). Dacă $B=0$, atunci automatul Mealy va trece într-o altă stare, $q_1$, generînd la ieşire *110*. Din această stare va trece în alta, $q_2$, în care în funcţie de $A$ va genera *001* sau *010* ş.a.m.d. Se obţine un automat Mealy cu patru stări, conform organigramei din figura 5.71.



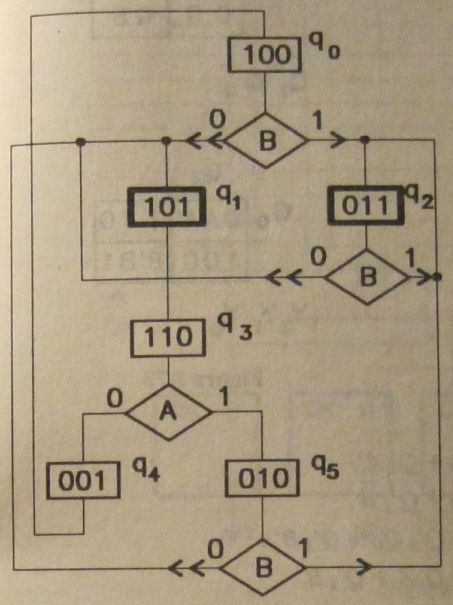



**Figura 5.70**

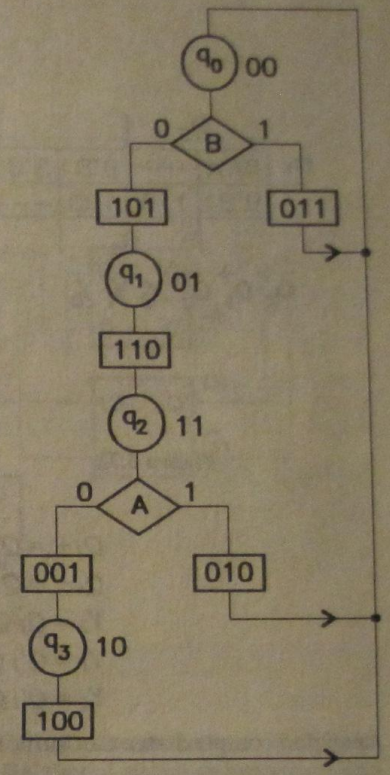



**Figura 5.71**

Am obţinut un automat cu un *CLC* mai simplu sau mai complicat? Putem răspunde la întrebare realizînd implementarea automatelor. Vom codifica stările automatului Moore astfel încît $Y \equiv Q$, iar stările celui de tip Mealy astfel încît tranziţiile condiţionate să se facă în stări codificate cu dependenţă redusă iar cele necondiţionate să presupună variaţie minimă. Rezultă diagramele de tranziţie din figura 5.72, pentru varianta Moore şi din figura 5.73, pentru varianta Mealy. În primul caz se obţin următoarele ecuaţii:

$$Q_2^+ = Q'_1 Q_0 + Q'_2 B' + Q'_1 B',$$
$$Q_1^+ = Q_2 Q_0 + Q_2 Q_1 A + Q_2 Q'_1 B + Q'_2 Q_1 B,$$
$$Q_0^+ = Q_2 Q'_1 Q'_0 + Q'_2 Q_1 + Q_1 A',$$
$$Y_2 = Q_2,$$
$$Y_1 = Q_1,$$
$$Y_0 = Q_0,$$

iar în cel de al doilea:

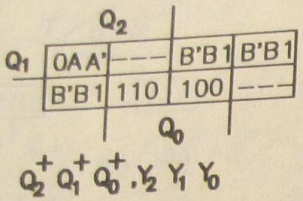



Figura 5.72

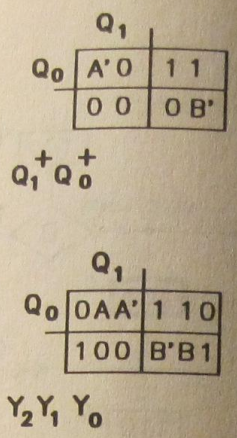



Figura 5.73

$$Q_1+ = Q'_1 Q_0 + Q_0 A',$$
$$Q_0+ = Q'_1 Q_0 + Q'_1 B',$$
$$Y_2 = Q_1 Q'_0 + Q'_1 Q_0 + Q'_1 B',$$
$$Y_1 = Q'_1 Q_0 + Q_0 A + Q'_1 B,$$
$$Y_0 = Q'_1 Q'_0 + Q_1 Q_0 A'.$$

Analizînd complexitatea *CLC*-urilor rezultă

$$S(CLC\ Moore) = 36,$$
$$S(CLC\ Mealy) = 31.$$

Deci, în funcţie de contextul oferit de aplicaţie, oricare din soluţii poate fi luată în consideraţie, conform comentariilor făcute cu ocazia *Exemplului 5.22*.

Vom desena circuitul în varianta Mealy cu întîrziere realizînd *CLC*-ul asociat cu o structură de tip *PLA* (figura 5.74). În seriile standard de circuite integrate există structuri de tip *PLA + Registru* ce permit implementarea automatelor finite într-o astfel de manieră. □

• **Frecvenţa maximă a ceasului** ce comandă un automat de tipul celor anterior proiectate se poate calcula luînd în consideraţie timpii de propagare prin *CLC*-uri şi timpii ce caracterizează comutarea bistabililor.

Pentru un *automat cu toate variabilele de intrare asincrone* se ia în calcul numai timpul de propagare pe buclă. Astfel, perioada ceasului, în cazul cel mai defavorabil, se determină astfel:

$$T_{min} = t_{pmaxCLC} + t_{SU} + t_{pD},$$



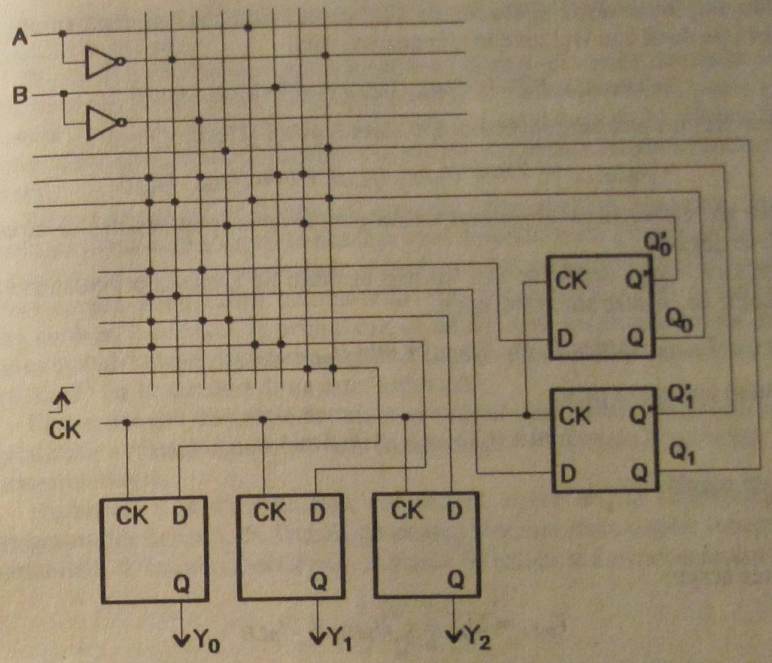



Figura 5.74

sumînd timpul maxim de propagare prin *CLC*-ul ce calculează starea următoare, timpul de set-up la intrarea bistabilului de tip *D* şi timpul maxim de propagare prin bistabilul de tip *D*. Acest calcul presupune că

$$t_H < t_{pD} + t_{pCLC}.$$

Se poate uşor arăta că această condiţie este sigur îndeplinită deoarece, pentru orice bistabil, timpul de hold este strict mai mic decît timpul de propagare minim, adică

$$t_H < t_{pDmin}.$$

Deci, bucla nu se poate "închide" într-un interval de timp mai mic decît $t_H$, nepermiţînd astfel funcţionarea automatelor finite.

Dacă *automatul are măcar o variabilă sincronă* va trebui luat în consideraţie şi timpul ei de propagare de la intrarea automatului la intrarea registrului de stare, astfel că:

$$T_{min} = max\ ((t_{pmaxCLC} + t_{SU} + t_{pD}),\ (t'_{pmaxCLC} + t_{SU} + t_S)),$$

unde: $t'_{pmaxCLC}$ este timpul de propagare maxim de la intrarea sincronă pînă la intrarea registrului de stare prin *CLC*-ul ce calculează starea următoare, iar $t_S$ este timpul de stabilire al variabilei de intrare drept consecinţă a frontului activ al ceasului.

În implementările realizate cu *CLC*-uri cu adîncime minimă se obţine o structură pe două sau trei niveluri inversoare deci:

$$t_{pmaxCLC} = max\,(t_{pHL}, t_{pLH}) + min\,(t_{pHL}, t_{pLH})$$

deoarece nu sînt necesare inversoare, bistabilii avînd şi ieşiri $Q'$;

$$t'_{pmaxCLC} = 2\,max\,(t_{pHL}, t_{pLH}) + min\,(t_{pHL}, t_{pLH})$$

deoarece pe intrări, în cazul general, există şi un inversor pentru calculul variabilei de intrare negate.
De regulă $t_S = t_{pD}$, deoarece sînt folosiţi bistabili de acelaşi tip pentru stocarea variabilelor de intrare sincrone, deci:

$$T_{min} \approx 2\,max\,(t_{pHL}, t_{pLH}) + min\,(t_{pHL}, t_{pLH}) + t_{SU} + t_{pD}.$$

Vom putea aproxima prin:

$$T_{min} \approx 3\,max\,(t_{pHL}, t_{pLH}) + 2\,min\,(t_{pHL}, t_{pLH})$$

şi cum, de regulă

$$t_{pHL} < t_{pLH}$$

vom putea scrie:

$$T_{min} \approx 3t_{pLH} + 2t_{pHL} < 5t_{pLH}.$$

Vom putea scrie deci acoperitor că

$$f_{max} \approx \frac{1}{5t_{pLH}}$$

putînd astfel *estima frecvenţa maximă* la care se poate construi un automat finit într-o tehnologie dată, pentru care cunoaştem timpul de propagare asociat tranziţiei pozitive, $t_{pLH}$. Spre exemplu, în tehnologia *TTL* standard, $t_{pLH} \approx 20ns$, deci vom putea construi automate finite care să funcţioneze la frecvenţe maxime în jur de *10 MHz*.

## 5.6 Automate cu reacţie negativă

### 5.6.1 Definiţie. Exemple

Comportamentul unui automat este caracterizat prin faptul că există configuraţii binare de intrare $X_i \in X$ astfel încît, prin menţinerea lor neschimbată la intrare, la ieşire este sesizată o evoluţie datorată unei traiectorii parcurse de automat în spaţiul stărilor. Deci, poate exista *un comportament la ieşire în absenţa*



*unei dinamici a intrării*. Acest fapt rezultă din *autonomia* circuitului faţă de comanda dată pe intrare.

Dar autonomia se mai poate manifesta şi prin *ignorarea dinamicii intrării*. Pentru anumite modificări ale configuraţiilor de intrare automatul poate rămîne "insensibil". Automatul poate oricînd evolua într-un subspaţiu al stărilor în care valoarea unor variabile de intrare nu este testată, caz în care dinamica acestora este ignorată.

Primul comportament are un caracter preponderent *generator*, deoarece presupune o dinamică a ieşirii în absenţa unei dinamici corespunzătoare a intrării. Cel de al doilea comportament tinde să *inhibe* comportamentul intrării, anulîndu-l ca efect asupra funcţionării automatului. Am putea vorbi, printr-o extensie de limbaj, credem, justificată, în primul caz, de un efect de *reacţie pozitivă* iar, în cel de al doilea caz, de un efect de *reacţie negativă*. În prima ipostază, dinamica intrării este "amplificată" iar în cea de a doua este "atenuată".

Facem această distincţie deoarece are efecte importante asupra structurării şi optimizării automatelor finite. Pentru o mai bună înţelegere a lucrurilor vom da trei exemple limită.

**Exemplul 5.28** Considerăm automatul strict iniţial reprezentat prin organigrama din figura 5.75. Tranziţiile acestui automat sînt complet independente de variabilele de intrare. Deci $X=\emptyset$, funcţiile de tranziţie ale automatului fiind

$$\lambda : Q \to Q$$
$$\delta : Q \to Y$$

(este lipsită de sens în acest caz distincţia Moore-Mealy). Sîntem în faţa unui *automat pur generator*, cu o evoluţie complet autonomă. Putem spune, în acest caz, că reacţia prin *CLC*-ul asociat este *pur pozitivă*. □

**Exemplul 5.29** Automatul descris de organigrama din figura 5.76 are o evoluţie ce depinde în fiecare stare de toate variabilele de intrare. În acest caz definiţia se aplică efectiv, fiecare element al produsului cartezian $(X \times Q)$ are asociat o stare distinctă. Tranziţia semnalului la intrare este maximum resimţită în evoluţia automatului. În fiecare stare sînt testaţi toţi biţii configuraţiei binare de intrare. Diversitatea evoluţiilor ieşirii este maximă. □

**Exemplul 5.30** Organigrama din figura 5.77 descrie un automat ale cărui tranziţii depind, de fiecare dată, numai de *un singur bit* din cei patru, ce formează configuraţia binară de intrare, *ABCD*. Funcţiile de tranziţie ale acestui automat vor putea fi puse sub următoarea formă

$$\lambda : P^*(X) \times Q \to Q$$
$$\eta : Q \to Y.$$

Într-adevăr, tranziţia din starea $q_1$, spre exemplu, se va face în $q_1$, dacă $X = A0CD$, sau în $q_3$, dacă $X = A1CD$. În starea $q_1$ mulţimea $X$ se împarte în două părţi, una a configuraţiilor binare pentru care $B=0$ şi alta a configuraţiilor binare în care $B=1$. Deci tranziţia se face condiţionat de părţi ale mulţimii $X$ şi nu de elemente ale acestei mulţimi.

De fiecare dată este eludată mare parte din diversitatea pe care o pot avea configuraţiile binare de intrare.

Acest caz, intermediar, merită să fie aprofundat. Diagramele de tranziţie sînt prezentate în figura 5.78. Din ele rezultă, codificînd stările cu biţii cei mai puţin semnificativi ai ieşirii (pentru că se poate în acest caz particular):

$$Q_1^+ = Q_1 Q_0 D + Q'_1 Q_0 B + Q'_1 Q'_0 C'$$
$$Q_0^+ = Q'_1 Q_0 + Q_0 D' + Q'_1 C' + Q_1 Q'_0 A'$$
$$Y_2 = Q_1 Q_0 + Q'_1 Q'_0$$
$$Y_1 = Q_1$$
$$Y_0 = Q_0$$

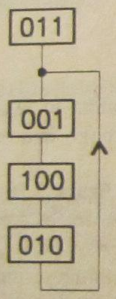



Figura 5.75

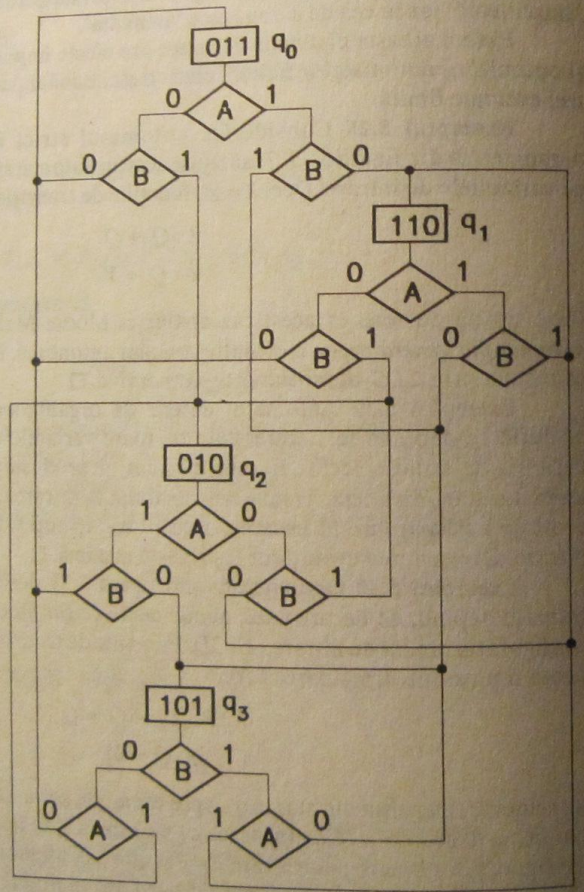



Figura 5.76



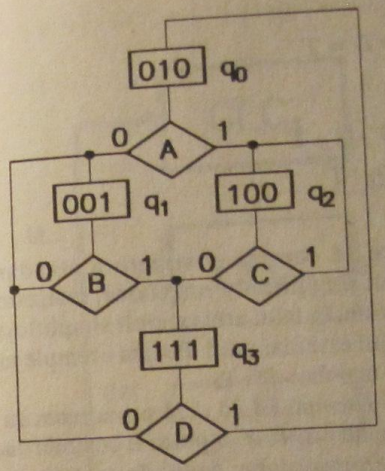



Figura 5.77

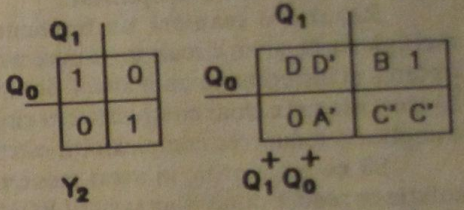



Figura 5.78

Să presupunem că putem găsi un procedeu prin care în fiecare stare este adus la intrarea automatului numai bitul ce trebuie testat. Astfel, în $q_0$, la unica intrare a automatului este conectat bitul $A$, în $q_1$ bitul $B$ ş.a.m.d. Acest lucru este posibil cu ajutorul unui multiplexor comandat cu biţii de stare, conform reprezentării din figura 5.79 unde este reprezentat semiautomatul ataşat acestui automat. În starea $q_0$, pentru $Q_1Q_0 = 10$, la intrarea $T$ a $CLC$ este conectată variabila $A$, în $q_1$, pentru $Q_1Q_0 = 01$, la intrarea $T$ este conectată variabila de intrare $B$ ş.a.m.d.

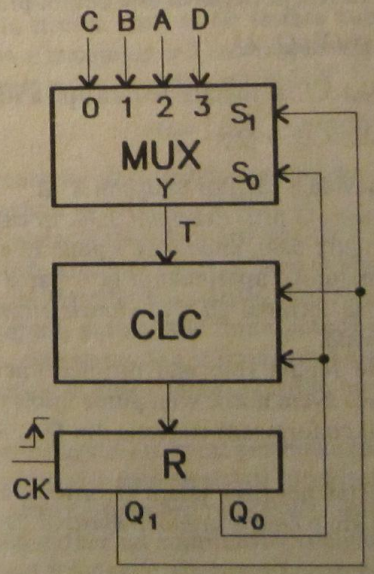



Figura 5.79

Notînd, în consecinţă

$$A = B = C = D = T$$

vom putea scrie

$$Q_1^+ = Q_0\,T + Q'_1\,Q'_0\,T'$$
$$Q_0^+ = Q'_1\,Q_0 + T'$$

obţinînd un *CLC* mult simplificat.

Rezultă o manieră de implementare ce conţine o structură standard, multiplexorul, şi un circuit logic oarecare mult simplificat. Proiectarea, realizarea, testarea şi întreţinerea unui astfel de circuit este, în felul acesta, mult simplificată. În acest exemplu chiar complexitatea circuitului este mai mică. Pentru exemple mai complicate reducerea complexităţii este mult mai drastică. □

Să ne reamintim, în acest moment, de *Exemplul 3.36* cînd promiteam că o soluţie ce reduce dimensiunea unui *ROM* folosind un *MUX* ar putea fi posibilă dacă am poseda codul de selecţie. Ne-am ţinut de promisiune: codul de selecţie este $Q_1Q_0$. *CLC*-ul automatului anterior poate fi un *ROM* de foarte mică dimensiune datorită *MUX*-ului ce-i comandă una din intrări.

Am putea, în consecinţă, nuanţa definiţia dată semiautomatului, asociat oricărui tip de automat, după cum urmează.

**Definiţia 5.12** Un *semiautomat pur generator* este definit prin

$$SAG = (Q, \lambda)$$

unde $\lambda$ este funcţia de tranziţie în spaţiul stărilor, $Q$, de forma

$$\lambda : Q \to Q\,. \;\square$$

**Definiţia 5.13** Un *semiautomat cu reacţie negativă* este definit prin

$$SAN = (X, Q, \lambda)$$

unde $X$ şi $Q$ au semnificaţia din *Definiţia 5.1* iar funcţia de tranziţie a stării este

$$\lambda : P^*(X) \times Q \to Q\,,$$

unde, prin $P^*(X)$ am notat mulţimea nevidă a părţilor mulţimii $X$. □

Această ultimă definiţie este ilustrată prin *Exemplul 5.30*. În cazul general structura unui *SAN* este ilustrată în figura 5.80, unde $CLC_1$ pune în evidenţă, în funcţie de $Q_-$ (deci de starea automatului), apartenenţa la o parte sau alta a mulţimii $X$. Acest circuit furnizează, la intrarea $CLC_2$, *numele elementului din* $P^*(X)$ *ce este semnificativ în starea curentă.*

În cazuri particulare limită, de regulă cele mai propice unei astfel de implementări, $CLC_1$ este un *MUX*. Dacă avem noroc vom putea folosi o parte sau toţi biţii lui $Q_+$ pentru a substitui biţii ce formează codurile din $Q_-$. Acesta este cazul *Exemplului 5.30*.

**Definiţia 5.14** Atunci cînd reacţia negativă poate fi introdusă printr-un multiplexor spunem că semiautomatul are o *reacţie negativă maximă*. □



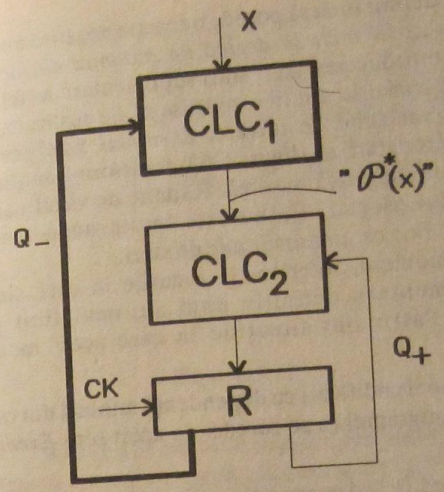



**Figura 5.80**

- *CLC*-ul asociat unui *SAN* corespunde unui *CLC* realizat cu *ROM* minimizat prin descompunere (vezi *Exemplul 3.36* din paragraful 3.4.1).
- *Întotdeauna un SA poate fi transformat într-un SAN cu reacţie negativă maximă prin introducerea de stări suplimentare.*

**Propoziţia 5.2** Dacă un semiautomat are $n$ stări atunci *SAN* cu reacţie negativă maximă asociat are maximum $n^2$-$n$ stări. □

*Demonstraţie.* Dacă din fiecare stare semiautomatul poate comuta condiţionat, în oricare altă stare, atunci organigrama va cuprinde, asociat fiecărei stări, un arbore de teste cu $\lceil log_2 n \rceil$ niveluri. Transformarea acestui arbore de teste, astfel încît în fiecare stare să fie testată numai o variabilă de intrare, presupune introducerea a maximum *n-2 stări suplimentare*. Acest lucru trebuie făcut pentru fiecare din cele $n$ stări de unde rezultă că

$$max\ |Q| = n(n-2) + n = n^2 - n.$$

În principiu, dacă semiautomatul are $O(n)$ stări, semiautomatul cu reacţie negativă maximă va avea $O(n^2)$ stări. □

Deci, această transformare într-un *SAN* cu reacţie negativă maximă se face dublînd biţii de stare, în cazul cel mai defavorabil (cu excepţia cazului limită cînd $n=2$), datorită introducerii stărilor suplimentare.

De regulă, automatele finite posedă din start o reacţie negativă. Problema care se pune este aceea de a le transforma astfel încît *reacţia negativă să fie maximă*. Din acest motiv, în majoritatea cazurilor, nu se ajunge la dublarea numărului biţilor de stare.

Reacţia negativă maximă permite uneori optimizarea structurii automatului, în sensul reducerii complexităţii circuitului combinaţional şi al realizării lui parţiale cu circuite standard (*MUX*-ul). Sînt şi situaţii în care introducerea acestei reacţii suplimentare nu simplifică structura automatului. Astfel, reacţia negativă maximă poate fi numai o alternativă ce trebuie luată în consideraţie.

• Un automat poate fi astfel definit încît să posede o reacţie negativă maximă sau *definiţia lui se poate modifica astfel încît să devină un automat din această categorie*. Modificarea presupune introducerea unor stări suplimentare astfel încît să se obţină o organigramă în care tranziţia din fiecare stare să fie realizată, prin testarea a cel mult unei singure variabile de intrare. *Exemplul 5.26* poate fi concludent şi în acest caz. Introducerea stării $q_4$ (figura 5.69) transformă automatul într-unul în care acţionează o reacţie negativă maximă. Rămîne de văzut dacă se produce o simplificare a *CLC*-ului asociat. Ea nu este întotdeauna teoretic garantată, mai ales în cazul automatelor cu un număr mic de stări.

• **Concluzionăm**, în acest moment, referitor la cazurile în care sîntem obligaţi să modificăm la implementare definiţia unui automat finit prin *introducerea stărilor suplimentare*. Patru sînt situaţiile în care acest mod de soluţionare se impune:

(1) imposibilitatea realizării unei codificări cu dependenţă minimă din cauza unei topologii neconvenabile a organigramei (a se revedea în acest sens *Exemplul 5.20*);

(2) imposibilitatea realizării codificării cu variaţie minimă pe o organigramă configurată neconvenabil (conform cu *Exemplul 5.24*);

(3) imposibilitatea testării mai multor variabile asincrone într-o singură stare (ca în *Exemplul 5.26*);

(4) necesitatea, ce uneori se impune, de a introduce reacţie negativă maximă în cazul unui automat ce are cel puţin o tranziţie determinată de două sau mai multe variabile (conform celor mai sus discutate).

Reamintim faptul, că introducerea unei stări suplimentare într-o definiţie dată este posibilă numai printr-o analiză ce ţine cont de modul de conectare al automatului în sistem.

## 5.6.2 CROM

Două sînt modurile de conectare ale unui automat într-un sistem. O primă posibilitate este aceea care are în vedere automatul ca *circuit de comandă*, caz în care ieşirile automatului sînt cuplate la intrările unui sistem digital oarecare, fără ca de la acesta din urmă automatul să primească un semnal. A doua variantă presupune folosirea automatului ca *circuit de control*, deoarece de la ieşirile sistemului digital comandat, o parte din semnal se întoarce la automat. În figura 5.81 este reprezentată această manieră de interconectare. *Automatul de control* este conectat cu exteriorul pe căile *In* şi *Out*, generează comenzile, *Com*, către sistemul digital controlat şi recepţionează de la acesta biţii, cu semnificaţie independent alocată, indicatori, *Ind*. Acest automat evoluează în funcţie de comanda primită pe *In* şi în funcţie de indicatorii ce caracterizează efectul comenzilor, *Com*, asupra sistemului digital. Schema de principiu este reprezentată în figura 5.82, unde *CLC*-ul este realizat cu un *ROM* sau cu *PLA*-ul ce-i poate fi asociat. Automatul este deci un controler cu *ROM*, de unde şi abrevierea prin *CROM*.



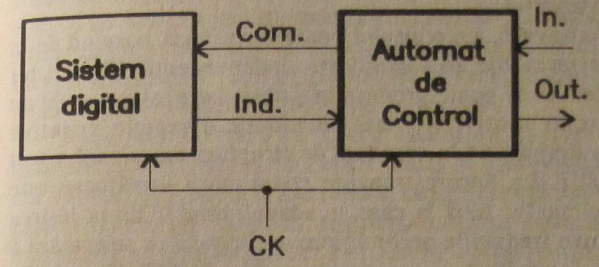



Figura 5.81

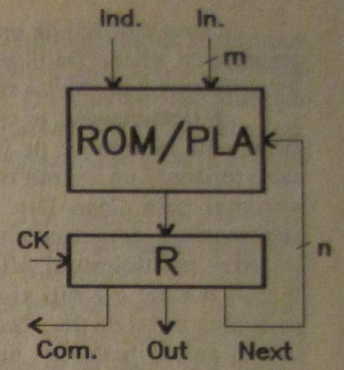



Figura 5.82

• Codul *In* permite *iniţializarea* automatului într-un număr de stări din care urmează evoluţia ce generează secvenţele de comenzi asociate unei funcţii oarecare realizată de ansamblul *sistem digital + automat de control*. O reducere a numărului de intrări în *ROM/PLA* se poate realiza prin structurarea reprezentată în figura 5.83, unde:

- *TC* este un transcodor ce translatează codul funcţiei, aplicat intrării *In*, în codul stării iniţiale asociat secvenţei de comenzi ce controlează funcţia;

- *MUX* este un multiplexor, care, de regulă, permite închiderea buclei pe care se propagă starea curentă (pentru *Mod=0*), iar după terminarea secvenţei asociate unei funcţii permite reiniţializarea automatului pentru o nouă funcţie, selectînd codul stării iniţiale de la ieşirea circuitului combinaţional *TC* prin *Mod=1*.

Evident, numărul de funcţii distincte comandate pe *In* este mai mic decît numărul de stări ale automatului, astfel încît este îndeplinită, în imensa majoritate a cazurilor, condiţia ca $n > m$. Astfel rezultă necesitatea circuitului *TC*. În acelaşi timp, printr-o proiectare atentă se poate ajunge în situaţia în care *TC* să nu conţină nici un circuit logic, o parte din intrările, selectate prin *Mod=1*, putînd fi conectate la valori logice fixe (zero sau unu).

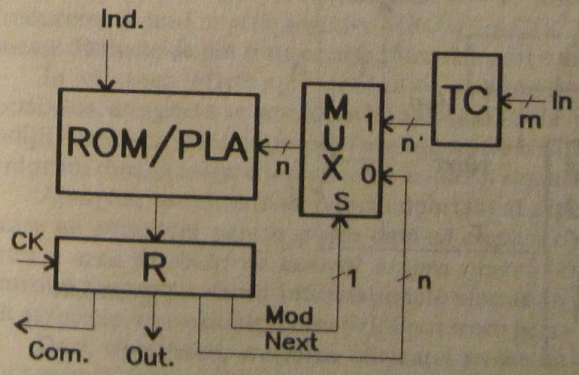



Figura 5.83

*Cîmpul* (field) de un bit *Mod* introduce o primă reacţie negativă, deoarece permite ca, uneori, biţii generaţi de *In* prin *TC* să poată fi ignoraţi. Cîmpul *Next* asigură reacţia pozitivă a sistemului.

• **Optimizarea**, în continuare, a circuitului poate fi realizată pornind de la observaţia că biţii de pe intrarea *Ind* au semnificaţie independentă. Fiecare bit caracterizează un anumit efect ce se poate produce în sistemul digital controlat de automat prin *Com*. Din acest motiv se poate introduce o reacţie negativă suplimentară, prezentată în figura 5.84 în care faţă de structura anterioară a fost introdus multiplexorul *MUXT*. La fiecare tranziţie codul stării următoare este format din cei *n-1* biţi ai cîmpului *Next* la care se adaugă bitul *T* de la ieşirea multiplexorului de test. Pentru tranziţiile necondiţionate cîmpul *Test* selectează *0* sau *1*, valori "cablate" pe intrările *0* şi, respectiv, *p* ale *MUXT*. Tranziţiile condiţionate se fac numai funcţie de un singur bit, cel selectat de *MUXT* dintre indicatorii ce vin pe calea *Ind*, şi sînt cuplaţi la intrările *1, 2, ..., p-1* ale *MUXT*.

Dacă *T* este bitul cel mai puţin semnificativ al codului de stare, spre exemplu, atunci stările ce urmează condiţionat unei stări oarecare sînt codificate printr-o pereche de numere binare ce diferă numai prin acest bit. Pentru *T=0* starea următoare va fi

$$XX \ldots X0$$

iar pentru *T=1* va fi

$$XX \ldots X1,$$

cei *n-1* biţi cei mai semnificativi (notaţi cu *X*) fiind aceiaşi în ambele cazuri. În *ROM*, comenzile, asociate acestor două stări, se vor afla localizate la adrese succesive.

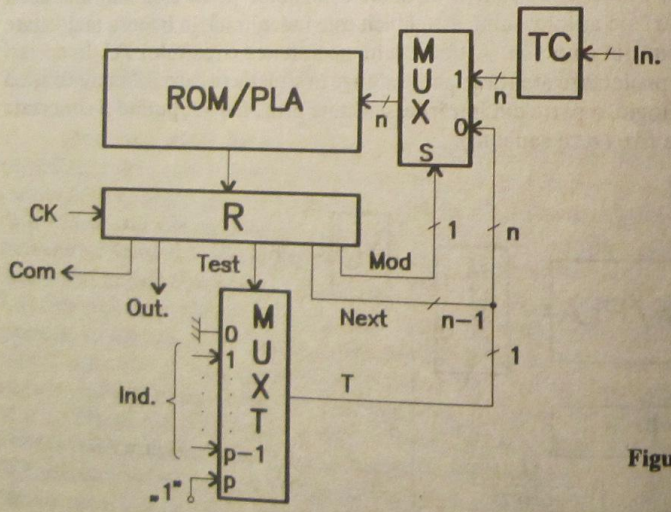



**Figura 5.84**



Astfel, în două etape, prin introducerea unor reacţii negative, de la intrarea *ROM*-ului (sau *PLA*-ului echivalent) au dispărut codurile binare asociate la *In* şi *Ind*, cu un efect copleşitor asupra complexităţii acestuia. Dacă, *In* sînt codificate pe *s* biţi iar indicatorii sînt în număr de *t*, atunci complexitatea *ROM*/*PLA*-ului scade de aproximativ $2^{s+t}$ ori. Cele două *MUX*-uri adăugate reprezintă structuri standard ce nu implică un efort în etapa de proiectare.

Singura limitare ce apare este dată de faptul că funcţia de control a automatului se exercită, în fiecare stare, în funcţie de cel mult un singur indicator, cel selectat prin *MUXT*. De regulă, imaginarea şi realizarea procesului de control este astfel cea mai comodă, restricţia anterior enunţată fiind tolerabilă. De altfel, este foarte greu de imaginat un proces de control care să presupună comenzi pentru a căror generare să fim obligaţi să ţinem cont simultan de valoarea mai multor indicatori. (Este greu de presupus că dacă am comandat o operaţie de adunare sistemului digital controlat, spre exemplu, sîntem interesaţi, pentru a evolua în continuare, dacă rezultatul a fost zero sau nu, dacă a dat sau nu carry operaţia executată sau dacă a rezultat din operaţie un număr pozitiv sau unul negativ. De regulă, într-un proces normal de gîndire a unui algoritm nu reuşim să fim interesaţi, într-o etapă decît de valoarea luată de un indicator. Sînt greu de imaginat algoritmi în care să fim interesaţi de testarea simultană a mai multor indicatori. Este vorba de o limită a modului nostru de gîndire, de o limită a formalismelor cu care am fost acomodaţi, de o limită, poate, a procesului educaţional care ne-a format.) În cazul în care, prin absurd, o astfel de configuraţie este excesiv de limitativă, se pot concepe automate de control cu două multiplexoare de test sau cu mai multe, caz în care se vor putea concepe şi tranziţii funcţie de doi sau mai mulţi indicatori. Schema va fi, în continuare, eficientă dacă numărul acestor multiplexoare nu se apropie de *t*. Experienţa arată că un singur *MUXT* este suficient în imensa majoritate a cazurilor.

• Instrumentul obţinut în această etapă este foarte puternic şi eficient datorită faptului că permite procese de control deosebit de ample utilizînd o structură cu o complexitate rezonabilă. *Complexitatea ROM-ului depinde polinomial de lungimea secvenţelor de comenzi* transmise către sistemul digital controlat. Varianta iniţială (figura 5.82) depindea *exponenţial* de numărul indicatorilor (*t*) şi de numărul de biţi cu care se codificau funcţiile (*s*), iar prin introducerea a două reacţii negative $S(ROM) = O(2^n)$, unde $2^n$ este numărul de comenzi distincte ce pot fi transmise către sistemul controlat.

În virtutea eficienţei dobîndite aplicaţiile *CROM*-ului s-au extins spectaculos, atingîndu-se complexităţi apreciabile ale *ROM*-ului asociat. În aceste condiţii, structura informaţională (configuraţiile binare stocate în *ROM*) necesită o ordonare care să asigure o *proiectare, testare şi depanare eficientă.*

Ajungem, astfel, într-un punct important al expunerii noastre, este punctul în care ne permitem pentru prima dată *să creştem complexitatea structurii fizice pentru a crea facilităţi de control asupra acurateţei structurii informaţionale.* Controlul asupra structurii informaţionale stocate în *ROM* devine atît de dificil încît aspectele legate de structura fizică pot trece pe un plan secund.

Deci, sintetizînd, creşterea eficienţei structurii fizice permite dezvoltarea unor structuri informaţionale complexe pentru al căror control sîntem obligaţi să

renunţăm la o maximă compactare a structurii fizice, după cum vom vedea în continuare.

• Structura din figura 5.84, maximum compactată structural, *impune* o anumită alocare restrictivă a codurilor asociate stărilor, după cum am văzut. Structura, ordonarea comenzilor în *ROM*, va fi astfel restricţionată de un mod impus de codificare a stărilor. Secvenţe de comenzi ce se succed în timp nu vor avea asociate şi o succesiune corespunzătoare în spaţiul adreselor *ROM*-ului. Vor rezulta de aici dificultăţi în proiectare, punere la punct şi mai ales în depanare. O structură informaţională complexă poate fi stăpînită uşor dacă este ordonată după criterii de conţinut şi nu după criterii ce nu-i sînt proprii, fiind impuse, spre exemplu, de criterii de codificare a stărilor sau de eficienţă a structurii fizice.

Propunem în acest sens configuraţia din figura 5.85, unde sînt specificate explicit cele două stări ce pot urma uneia din care tranziţia se produce condiţionat. Cîmpurile *Next 0* şi *Next 1* generează stările următoare pentru $T=0$ şi, respectiv, $T=1$ dacă $Mod=1$, altfel, pentru $Mod=0$ se poate iniţializa sistemul, la începutul unei noi secvenţe de comenzi exterior impusă de codul *In*, pentru $T=0$, sau în starea *00...0*, dacă $T=1$ (*resetarea CROM*-ului). Valoarea bitului *T* poate fi selectată, prin cîmpul *Test*, dintre valorile curente ale indicatorilor (*Ind*) sau dintre cele două valori fixe *0* şi *1* conectate la intrările *0* şi, respectiv, *p* ale *MUXT*.

Ne vom putea permite să organizăm acum conţinutul *ROM*-ului astfel încît să reflecte mai limpede diferite secvenţe de comenzi ce vor putea fi scrise compact, la adrese succesive, pentru a putea fi proiectate şi testate comod. Nu este vorba de o comoditate subiectiv condiţionată, ci este vorba de crearea premizelor parcurgerii rapide a unor etape ce premerg lansării unui produs. Flexibilitatea funcţională a *CROM*-ului cîştigă foarte mult în condiţii în care tocmai pe această flexibilitate se bazează opţiunea pentru folosirea unei astfel de structuri într-un sistem digital complex.

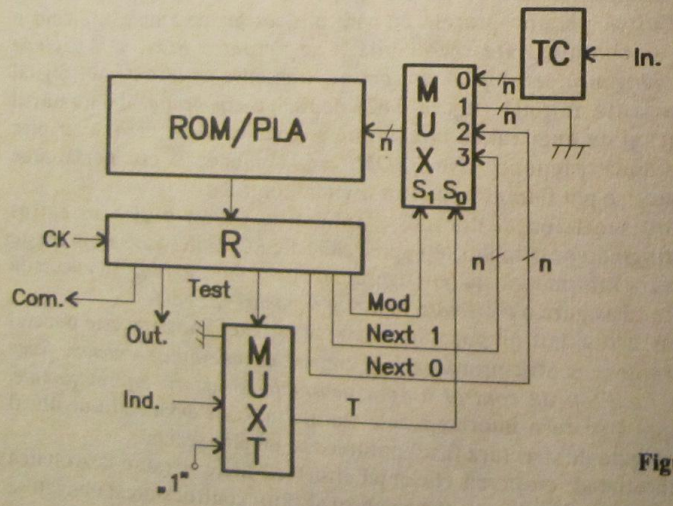



**Figura 5.85**



• **Varianta de tip Moore imediat** a *CROM*-ului, anterior prezentat, este dată în figura 5.86. Din punct de vedere funcţional nu apar diferenţe. Numai din punctul de vedere al realizării practice, aceste diferenţe sînt importante. Registrul asociat se micşorează sensibil, fapt ce trebuie marcat ca fiind strict pozitiv. Ieşirile *Com* şi *Out* sînt generate direct dintr-un *CLC* fiind, în consecinţă, afectate de hazard. După cum este lesne de înţeles nu ne vom permite în cazul *CROM*-ului restricţii de codificare ce ar putea elimina hazardul pe care va trebui să-l acceptăm aici, datorită complexităţii, ca pe o fatalitate. De multe ori putem astfel aranja lucrurile ca acest hazard să nu afecteze buna funcţionare a sistemului.

Adăugăm celor anterioare, modificarea relaţiei între ceasul sistemului şi timingul asociat semnalelor *Com*, *Out*, *Ind*. Noua relaţie temporală poate fi uneori utilă. Reprezintă, oricum, o alternativă de care vom ţine cont în secţiunile următoare, cînd se va pune problema vitezei de lucru a unor sisteme mai complex structurate.

• Introducerea a două cîmpuri pentru starea următoare, *Next 0* şi *Next 1*, măreşte dimensiunea *ROM*-ului în limite acceptabile. La fel, apariţia cîmpurilor *Mod* şi *Test* ce au permis introducerea reacţiilor negative. Este vorba însă de o mărire ce afectează cu o constantă aprecierea complexităţii sistemului, ce rămîne în continuare $O(2^n)$, conform cu *Propoziţia 3.19*.

Ne vom putea permite însă o tentativă de optimizare a sistemului prin structura prezentată în figura 5.87 în care:

- *+1* este un circuit de incrementare ce calculează, pornind de la adresa (starea) curentă, pe cea strict următoare;

- $TC_1$ este un transcodor ce permite introducerea unor modalităţi mai complexe de tranziţie a sistemului;

- intrările *MUXT* vor fi folosite integral pentru indicatorii recepţionaţi de la sistemul digital controlat.

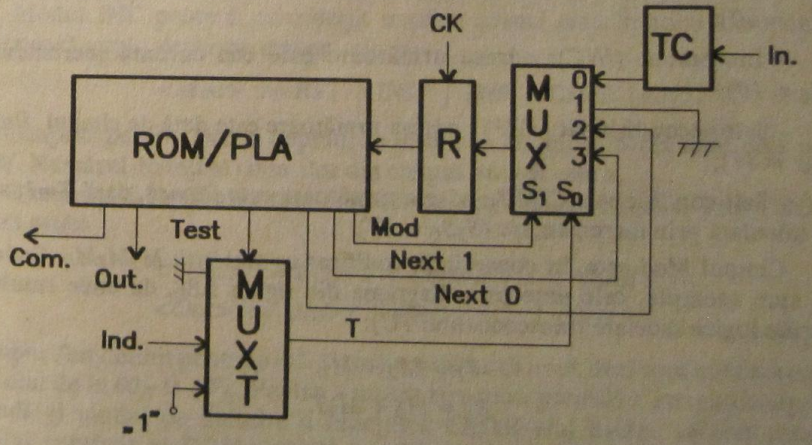



**Figura 5.86**

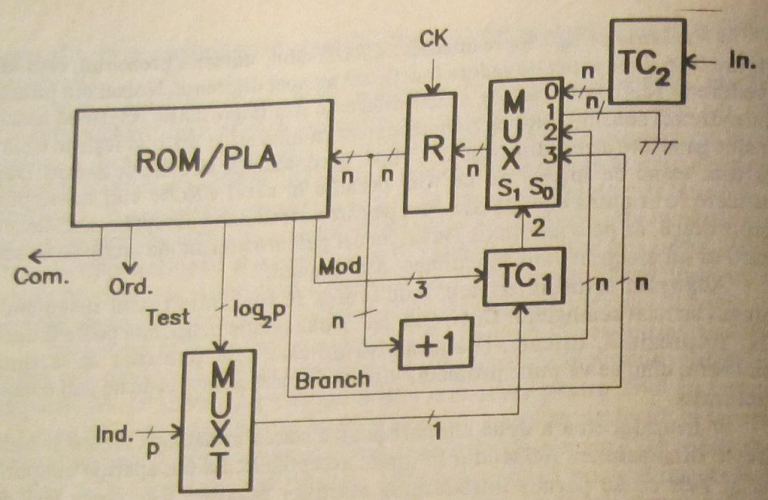



Figura 5.87

Evoluţia acestui automat se face calculînd stările următoare prin incrementare, dacă $T=0$ sau dacă tranziţia este necondiţionată, sau prin saltul, condiţionat ($T=1$) sau necondiţionat, la o altă ramură în arborele de comenzi, conform codului (adresei) generat de cîmpul *Branch*. *Next 0* se va calcula tot timpul prin increment, iar *Next 1* va deveni *Branch*. Astfel, printr-un sumator binar de $n$ biti, vom înlocui un *ROM* de $2^n$ locaţii de cîte $n$ biţi.

Modurile de funcţionare ale automatului sînt următoarele:

- Reset (*RST*): salt la adresa $00\ldots0$ ($S_1S_0 = 01$);

- Iniţializare (*INIT*): salt la începutul unei noi secvenţe conform intrării *In* ($S_1S_0 = 00$);

- Increment (*INC*): adresa următoare este cea curentă incrementată ($S_1S_0 = 10$);

- Salt necondiţionat (*JMP*) : adresa următoare este dată de cîmpul *Branch* ($S_1S_0 = 11$);

- Salt condiţionat (*CJMP*): adresa următoare este *Branch*, dacă $T=1$, altfel este calculată prin incrementare ($S_1S_0 = 1T$).

Cîmpul *Mod* este, în consecinţă, codificat cu trei biţi: $M_2M_1M_0$. Codurile sînt, spre exemplu, cele date prin diagrama din figura 5.88, de unde rezultă şi funcţiile logice asociate transcodorului $TC_1$:

$$S_1 = M_2 + M_1,$$
$$S_0 = M_0 + M_2T.$$



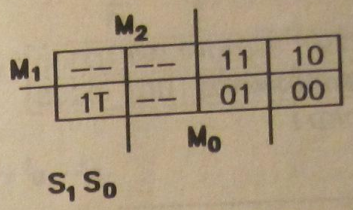



Figura 5.88

(Simplitatea lor este datorată şi modului abil în care s-a făcut alocarea codurilor. Cititorul poate încerca să afle în ce a constat această abilitate şi dacă nu există o soluţie şi mai simplă.)

• *CROM*-ul a fost structurat, prin etapele anterior parcurse, astfel încît s-au *segregat* o serie de funcţiuni realizabile cu circuite standard (*MUX*, sumator) sau configuraţii foarte simple ($TC_1$ sau $TC_2$) ce nu ţin de natura problemelor rezolvate. Introducerea structurii $+1$ a permis recompactarea *ROM*-ului cu preţul unui circuit standard simplu.

• Modul în care a fost definit *CROM*-ul, în stadiul cel mai avansat de segregare a funcţiilor ce-i controlează evoluţia, permite şi introducerea unui formalism simplu ce-i descrie funcţionarea.

Vom asocia fiecărei stări o *linie de comandă* definită după cum urmează:

*<Linie de comandă>* ::=*<Com><Out><Mod><Test><Branch>*

unde cîmpurile *<Com>*, *<Out>* şi *<Test>* se definesc concret în funcţie de sistemul digital controlat, iar celelalte astfel:

*<Mod>* ::= *RST* | *INIT* | *INC* | *JMP* | *CJMP*,
*<Test>* ::= *IND 0* | *IND 1* | *IND 2* | ... | *IND*($2^p$-1),
*<Branch>* ::= $00\ldots0$ | $00\ldots1$ | ... | $11\ldots1$.

Modul *INC* poate fi considerat implicit atunci cînd în acest cîmp nu se completează nimic, deci este mai practic ca:

*<Mod>* ::= *RST* | *INIT* | *JMP* | *CJMP* |–.

Semnificaţiile pentru *IND i* depind de sistemul digital concret controlat prin *CROM*. Numărul de biţi al codurilor din cîmpul *Branch* este $n$.

**Exemplul 5.31** Considerăm o structură oarecare, comandată prin *CROM*, definită astfel

*<Com>* ::= *COM 1* | *COM 2* | ... | *COM 9* |–,
*<Out>* ::= *ORD 1* | *ORD 2* | ... | *ORD 7* | –,

iar cîmpul *Test* definit pentru $p=3$. Vom presupune că $n=8$, deci vom nota adresele *ROM*-ului de la $00_H$ la $FF_H$. Pentru a ilustra structura posibilă a unor secvenţe de comandă şi modul de definire a funcţiilor *CROM*-ului listăm, în continuare, posibilul conţinut al *ROM*-ului, de la adresa $00_H$ la $0D_H$, pentru o aplicaţie ipotetică. Coloana din stînga reprezintă adresa *ROM*-ului:

$00_H$ *COM 5*
$01_H$ *COM 2 CJMP IND 4 00000011*
$02_H$ *COM 4 JMP 00000001*
$03_H$ *COM 2 ORD 1*
$04_H$ *INIT*

---

$05_H$ *COM 7 CJMP IND 1 00001000*
$06_H$ *COM ORD 7*
$07_H$ *COM 2*
$08_H$ *ORD 2 INIT*

---

$09_H$ *ORD 5 COM 3*
$0A_H$ *CJMP IND 6 00001100*
$0B_H$ *ORD 6 INIT*
$0C_H$ *COM 4 JMP 00001001*

---

$0D_H$ *RST*

Să lecturăm secvenţele anterioare! Prin iniţializarea *CROM*-ului acesta trece în starea *00000000* sau, altfel spus, citeşte linia de comandă de la adresa $00_H$ unde se află *COM 5* ce este dată sistemului digital controlat. În starea următoare se trece necondiţionat, prin incrementarea adresei, comandă implicit presupusă în linia $00_H$. Linia de la adresa $01_H$ (din starea *00000001*) conţine *COM 2* iar automatul va trece în starea *00000011* dacă *IND 4 = 1* sau în starea *00000010* (adresa $02_H$) dacă *IND 4 = 0*. În starea următoare, la adresa $02_H$, este generată comanda *COM 4* şi se sare înapoi, necondiţionat la adresa $01_H$. Starea următoare generează *COM 2* şi *ORD 1* şi determină tranziţia prin *INC*, presupus implicit. În starea $04_H$ se dă comanda *INIT* prin care automatul *CROM* este resetat într-o stare de început a unei secvenţe ce corespunde funcţiei selectate prin codul *In*.

În $05_H$ se află prima stare a unei alte funcţii controlate prin *CROM*, la fel în $09_H$ şi $0D_H$. Funcţia din $0D_H$ este funcţia de iniţializare a *CROM*-ului comandată printr-unul din codurile de intrare, *In*. De regulă, toate secvenţele trebuie să conţină cel puţin într-o linie de comandă modul *INIT* pentru a permite evoluţia *CROM*-ului şi sub comanda intrării *In*. Cînd această condiţie nu este îndeplinită poate fi suplinit *INIT* prin *RST* sau sistemul nu mai răspunde comenzilor din exterior.

*CROM*-ul proiectat are definite patru funcţii, controlate prin secvenţele iniţiate în $00_H$, $05_H$, $09_H$ şi $0D_H$. Vom nota aceste funcţii cu *FUNC 0, ... , FUNC 3* şi le vom codifica cu *00, 01, 10* şi, respectiv, *11*, cîmpul *In* fiind, în consecinţă, codificat cu $F_1F_0$. Funcţia logică a $TC_2$ va fi definită în diagrama din figura 5.89, de unde extragem



| $F_0$ \ $F_1$ | 0000 1101 | 0000 0101 |
|---|---|---|
| | 0000 1001 | 0000 0000 |

**Figura 5.89**

$I_7\ I_6 \ldots I_0$

$$I_7 = I_6 = I_5 = I_4 = 0,$$
$$I_3 = F_1,$$
$$I_2 = F_0,$$
$$I_1 = 0,$$
$$I_0 = F_1 + F_0,$$

unde prin $I_7, \ldots, I_0$ am notat ieşirile transcodorului $TC_2$ care generează configuraţia binară ce iniţializează automatul *CROM*. Dacă se definesc noi funcţii, atunci structura circuitului $TC_2$ se va modifica în consecinţă. □

Ne aflăm în faţa unui nou mod de definire, sau reprezentare, a unui automat finit. Acest mod a fost impus de complexitatea foarte mare a structurii. Descrierea prin organigrame sau alte procedee similare ar fi ineficientă în cazul *CROM*-urilor.

• **Translatarea liniei de comandă** în conţinutul binar al *ROM*-ului se face printr-un program care primeşte la intrare secvenţe de linii de comandă şi generează secvenţe binare corespunzătoare.

**Exemplul 5.32** Structura liniei de comandă, altfel spus, a cuvîntului din memoria *ROM*, este dată în figura 5.90. Numărul de biţi al fiecăruia din cele 5 cîmpuri a fost stabilit conform definiţiilor anterioare. Astfel, cîmpul *COM* are patru biţi pentru că există 9 comenzi efective la care se adaugă comanda *noop*, ce nu generează nici un efect în sistemul controlat şi este codificată prin *0000*. Codurile au fost asociate în funcţie de indicele *mnemonicei*. Spre exemplu, mnemonica *ORD 3* are codul *011*, sau *IND 5* din *TEST* are codul *101* ş.a.m.d. De regulă, cînd un cîmp nu conţine o valoare semnificativă, atunci el se umple cu zerouri.

În consecinţă, primele 5 linii, din *ROM*-ul asociat structurii din exemplul anterior, au următorul conţinut:

$00_H$ *0101.000.010.000.00000000*
$01_H$ *0010.000.100.100.00000011*

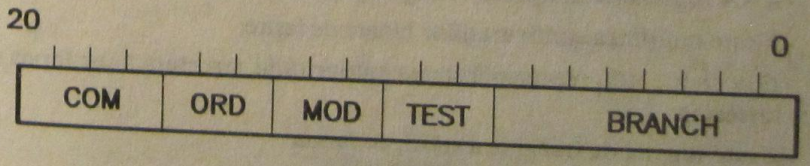



**Figura 5.90**

$02_H$ *0100.000.011.000.00000001*
$03_H$ *0010.001.010.000.00000000*
$04_H$ *0000.000.000.000.00000000.*

Codurile pentru cîmpul *MOD* au fost extrase din figura 5.88. Punctele delimitează cîmpurile liniei de comandă (pentru a uşura citirea). □

• **Cînd trecem de la ROM la PLA?** Nu ne putem gîndi la această tranziţie decît după ce proiectul din care *CROM*-ul face parte a fost finalizat printr-un prototip funcţional. O vom face întotdeauna cînd *CROM*-ul este realizat într-un singur circuit integrat, deducînd din *ROM*-ul, ce a fost verificat, structura *PLA*-ului echivalent. Mai putem adopta această soluţie şi în cazul în care *CROM*-ul este realizat cu mai multe capsule de circuite integrate şi *PLA*-ul rezultat este accesibil ca o componentă de catalog. Dacă nu, rămînem la soluţia cu *ROM*.

## 5.7 Automate cu spaţiul stărilor structurat ($AS^3$)

*CLC*-ul asociat unui automat finit primeşte la borne biţii de stare, $Q$, şi pe cei ai configuraţiei binare de intrare, $X$. Reacţia negativă a permis reducerea efectului biţilor de intrare asupra complexităţii *CLC*-ului. Structurarea spaţiului stărilor, de care ne vom ocupa în acest subcapitol, va reduce efectul biţilor de stare asupra complexităţii *CLC*-ului.

### 5.7.1 Definiţie. Efecte

Introducerea conceptului de *automat cu spaţiul stărilor structurat* ($AS^3$) se impune atunci cînd mulţimea $Q$ are o complexitate ce depăşeşte limite tehnologic acceptabile într-un anumit context.

**Definiţia 5.15** Un automat cu spaţiul stărilor structurat este definit prin următorul qvintuplu

$$AS^3 = (X \times A, Y, Q_1 \times Q_2 \times \ldots \times Q_q, \lambda, \delta)$$

unde:

- $X \times A$ reprezintă mulţimea configuraţiilor binare de intrare;
- $Y$ este mulţimea configuraţiilor binare de ieşire;
- $Q_1 \times Q_2 \times \ldots \times Q_q$ reprezintă starea automatului, structurată sub forma unui produs cartezian;
- $\lambda$ este functia de tranziţie a stării, de forma



$$\lambda : X \times Q_i \to Q_i$$

unde indicele $i$ este dat de valoare binară curentă aplicată intrării $A$;

- $\delta$ este funcţia de tranziţie a ieşirii, de forma

$$\delta : X \times Q_i \to Y$$

indicele $i$ fiind acelaşi ca în cazul funcţiei $\lambda$. □

Din definiţia anterioară rezultă că starea automatului se poate complet modifica în minimum $q$ cicluri de ceas, avînd drept consecinţă o diminuare spectaculoasă a complexităţii *CLC*-ului asociat. Elementul $A$ al produsului cartezian $X \times A$ reprezintă mulţimea prin care "adresăm" elementul produsului $Q_1 \times \ldots \times Q_q$.

**Propoziţia 5.13** $AS^3$ (figura 5.91*b*) asociat unui automat $A$ (figura 5.91*a*) cu $|Q| = 2^n$ are o complexitate $S_{AS^3}(n) = O(n \log n)$ faţă de $S_A(n) = O(2^n)$ a automatului cu spaţiul stărilor nestructurat. Produsul $S(n) \cdot T(n)$ scade de la $O(2^n)$, pentru $A$, la $O(n^2 \log n)$, pentru $AS^3$. □

*Demonstraţia* porneşte de la faptul că un multiplexor are complexitatea $S_{MUX}(n) = O(n \log n)$, unde $n$ este numărul de intrări selectate. Dacă fiecare element al produsului cartezian este o mulţime de configuraţii binare de $m$ biţi (în ipoteza, ce nu este excesiv de simplificatoare, că mulţimile $Q_i$ sînt identice) atunci bucla automatului $AS^3$ se închide prin $m$ *MUX*-uri de $n/m$ căi fiecare, conform reprezentării din figura 5.91*b*, şi un circuit combinaţional *CLC'* de complexitate $O(1)$. Starea automatului $A$ comută într-un tact, deci $T_A(n) = O(1)$, indiferent de $n$, iar cea a $AS^3$ va fi complet modificată în $n/m$ tacturi, deci $T_{AS^3}(n) = O(n)$. □

În figura 5.91*b* nu am prezentat o soluţie pentru distribuţia ceasului către subregistrele ce formează pe $R$. Fiecare $Q_i$ este stocat într-un subregistru, $R_i$, ceasul

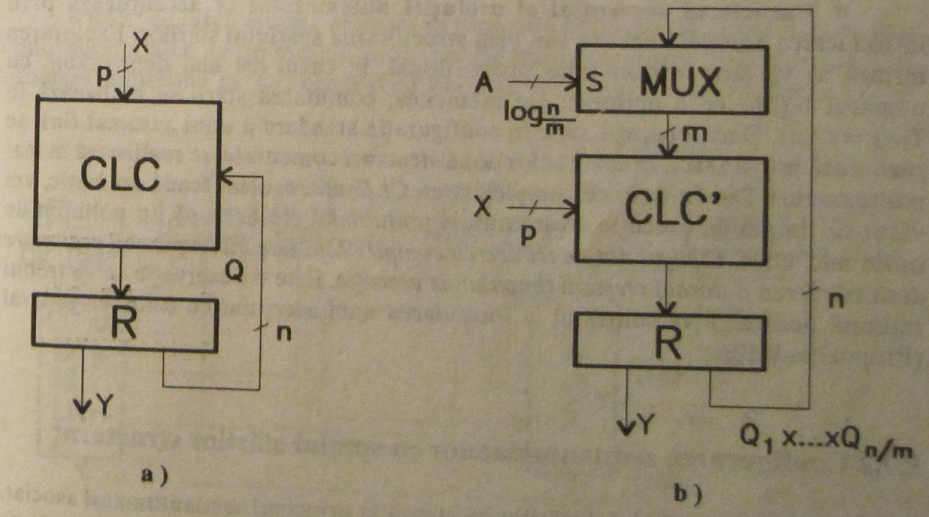
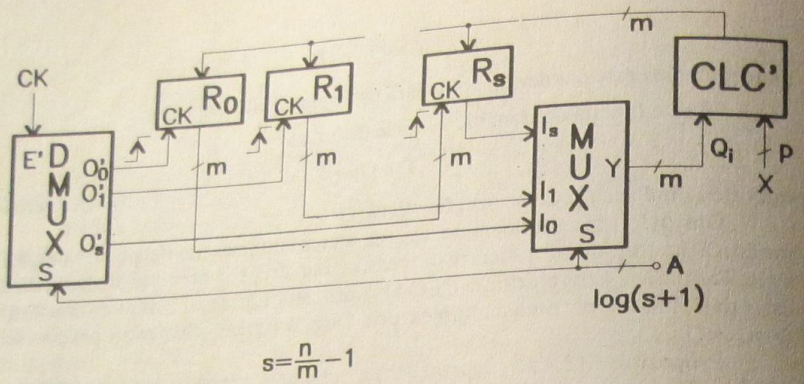



**Figura 5.91**




$$s=\frac{n}{m}-1$$

**Figura 5.92**

fiind aplicat, în funcţie de codul *A*, subregistrului ce conţine elementul produsului cartezian afectat de comutare. Figura 5.92 este explicită în acest sens.

• **Simetria între un AS$^3$ şi un semiautomat cu reacţie negativă** este evidentă. În ambele cazuri *multiplexăm* biţi ai produsului $Q\times X$. Într-un caz, pe cei din $Q$ (pentru $AS^3$) iar într-altul, pe cei din $X$ (pentru *SARN*). Pentru $AS^3$ multiplexorul este introdus pe buclă şi este comandat (pe intrările de selecţie) cu un cod de intrare ($A$) iar în cazul automatelor cu reacţie negativă, multiplexorul este conectat pe intrare şi este comandat printr-o parte a biţilor de stare ($Q_$). Simetria este evidentă şi dacă se analizează figurile 5.91*b* şi 5.80. În ambele cazuri este afectată, în principal, structura de semiautomat a automatului, în sensul reducerii drastice a complexităţii structurale.

• **Caracterul secvenţial al evoluţiei automatului se accentuează** prin introducerea reacţiei negative sau prin structurarea spaţiului stărilor. Explorarea intrării se va face într-un timp proporţional, în cazul cel mai defavorabil, cu numărul biţilor ce o definesc. De asemenea, comutarea stării se realizează în $T(n) = O(n)$. Deci, operaţii, care în configuraţia standard a unui automat finit se realizează într-un tact, în cazul celor două structuri comentate, se realizează în mai multe tacturi. Dar în timp ce complexitatea *CLC*-ului asociat scade dramatic, am văzut că timpul de execuţie creşte numai polinomial (chiar după un polinom de ordin mic, unu). *Cîştigul dat de reducerea complexităţii este incomparabil mai mare decît pierderea datorată creşterii timpului de execuţie*. Este o observaţie ce va trebui reţinută pentru că va contribui la formularea unei aserţiuni cu caracter general (Propoziţia 8.10).

## 5.7.2 Configurarea semiautomatelor cu spaţiul stărilor structurat

Structurarea spaţiului stărilor afectează în principal semiautomatul asociat. Din acest motiv ne vom ocupa de semiautomatul cu spaţiul stărilor structurat



($SAS^3$) punînd în evidenţă reconfigurările majore ce se impun atunci cînd se doreşte optimizarea structurii fizice.

• **Structura detaliată a unui SAS$^3$** este reprezentată în figura 5.92 unde, faţă de reprezentarea din figura 5.91*b*:

- $R_0$, $R_1$,..., $R_s$ (unde $s = (n/m) - 1$) se substituie registrului $R$ în care este stocată starea automatului;

- *DMUX*-ul distribuie ceasul, *CK*, către registrele $R_i$, unul din ele fiind selectat prin codul $A$ spre a fi modificat în ciclul curent;

- *MUX*-ul selectează elementul $Q_i$ din registrul $R_i$ conform adresei $A$ pentru a fi aplicat la intrarea *CLC'*;

- rezultatul ieşirii *CLC'* este înscris în $R_i$, singurul ce primeşte impuls de ceas în ciclul curent.

În fiecare ciclu de ceas se configurează un semiautomat format din $R_i$ şi *CLC'*, indicele $i$ fiind dat de codul $A$. $SAS^3$ este de fapt format din $s+1$ semiautomate ce funcţionează restricţionat de faptul că în fiecare ciclu de ceas comută numai unul singur. Circuitul *CLC'* este partajat în timp între cele $s+1$ semiautomate.

Registrele $R_i$, reamintim că sînt structuri de tip master-slave, realizate prin conectarea în cascadă a două latch-uri, master-ul şi slave-ul. Slave-ul realizează *funcţia de stocare* propriuzisă iar master-ul asigură netransparenţa buclei de ordinul doi pe durata palierului dinaintea frontului activ al ceasului. În fiecare ciclu de ceas este utilă funcţia de stocare a celor $s+1$ latch-uri slave şi funcţia de memorare temporară a latch-ului master din registrul $R_i$ numai. În fiecare ciclu, $s$ latch-uri master nu au nici un rost, de fiecare dată, unul din ele, numai, fiind cel ce asigură închiderea netransparentă a buclei. Pornind de la această observaţie se poate realiza o simplificare suplimentară.

• **SAS$^3$ cu latch master partajat în timp** este obtinut prin punerea în comun a unui singur latch de tip master, ca în reprezentarea din figura 5.93 unde *LATCH* formează împreună cu $L_i$ registrul ce permite de fiecare dată închiderea buclei

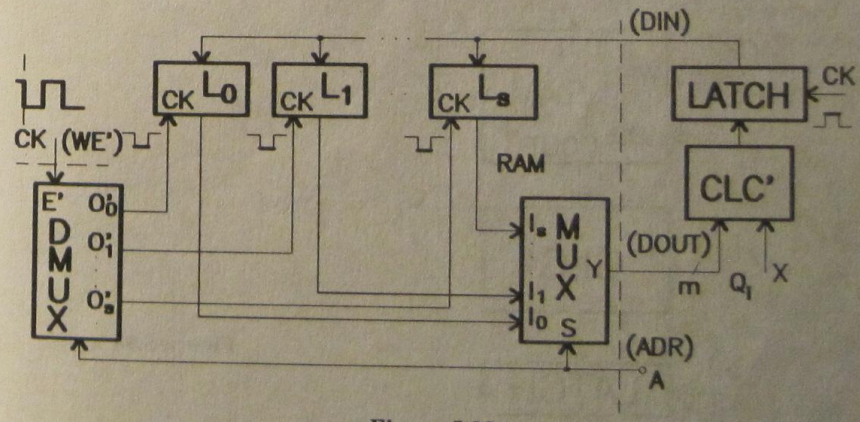



**Figura 5.93**

semiautomatului. Structura obţinută este mai simplă pentru că în loc de $R_i$ avem $L_i$. Latch-uri de $m$ biţi înlocuiesc registre de $m$ biţi. Deci în loc de $2(s+1)$ latch-uri, noua structură conţine $s+2$ latch-uri.

În prima parte a ciclului de ceas, $CK=1$, latch-ul *LATCH* este transparent. Conţinutul latch-ului $L_i$ (cu $i$ dat de codul $A$) împreună cu intrarea $X$ determină, prin *CLC'*, o configuraţie binară, care se va înscrie în *LATCH*. După tranziţia negativă a ceasului, în partea a doua a ciclului de ceas cînd $CK=0$, latch-ul *LATCH* se închide iar conţinutul său se "varsă" în $L_i$, care este acum transparent. Conţinutul celorlalte latch-uri slave rămîne nemodificat.

Spre deosebire de prima variantă, în cea de a doua, $SAS^3$ este astfel gîndit încît să fie partajate în timp două resurse: *CLC'* şi *LATCH*.

• **A treia variantă de SAS$^3$** se configurează pornind de la observaţia că $L_0$, $L_1$, ..., $L_s$, *DMUX*-ul ce distribuie ceasul *CK* şi *MUX*-ul ce selectează cîte un $Q_i$ din $L_i$ formează o structură de *RAM* cu $s+1$ locaţii (adrese) de cîte $m$ biţi fiecare. Intrările în $L_0$, $L_1$, ..., $L_s$ formează *DIN*, ieşirea *MUX*-ului este *DOUT*, codul de selecţie aplicat la *DMUX* şi *MUX* formează adresa *ADR*, iar intrarea *CK* aplicată *DMUX*-ului se poate asimila cu *WE'*. Vom redesena structura ca în figura 5.94 în care se pune în evidenţă structura maxim compactată a unui $SAS^3$, într-o variantă minimală din punctul de vedere al facilităţilor. În fiecare ciclu, conţinutul unei locaţii de memorie, specificată prin *A*, este prelucrat în *CLC'* împreună cu intrarea *X* iar rezultatul se reînscrie la aceeaşi locaţie de memorie. Prima fază a ceasului, $CK=1$, serveşte citirii din *RAM*, propagării prin *CLC'* şi înscrierii în *LATCH*. Cea de a doua fază a ceasului, $CK=0$, permite înscrierea conţinutului *LATCH*-ului la adresa *A*, de unde în prima fază a avut loc citirea. *LATCH*-ul, împreună cu locaţia de memorie selectată prin *A*, formează o structură master-slave ce permite funcţionarea corectă a structurii ca automat finit.

Prin partajarea circuitului combinaţional şi a *LATCH*-ului a fost posibilă *segregarea memoriei RAM* în structura unui $SAS^3$.

• **Ieşirea AS$^3$ asociat SAS$^3$** se poate realiza pe mai multe căi: de la ieşirea *RAM*-ului, de la ieşirea *CLC'* sau de la ieşirea *LATCH*-ului. Configuraţia de sistem

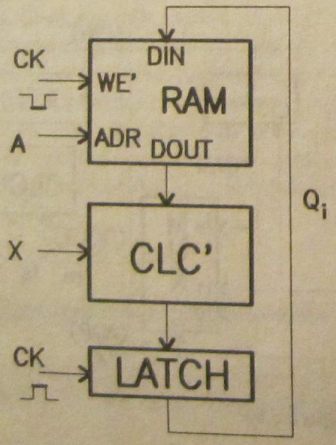



**Figura 5.94**



în care este introdus automatul va determina un mod de conectare sau altul. Se pot imagina şi mai multe moduri de extragere a ieşirii.

• **Segregarea RAM-ului** se adaugă efectelor evidenţiate prin *Propoziţia 5.13* impunînd structura de $AS^3$ ca una deosebit de eficientă în *realizarea prelucrării unor structuri de date complexe*.

• **AS$^3$ reprezintă o generalizare a conceptului de automat finit**, la care se reduce atunci cînd *RAM*-ul are o singură locaţie.

## 5.7.3 RALU

Una din aplicaţiile cele mai răspîndite ale $AS^3$ o constituie structura de *RALU* (*R*egisters with *A*rithmetic-*L*ogic *U*nit). O astfel de unitate se regăseşte, în diferite variante, în structura oricărui procesor standard, rolul ei fiind acela de prelucrare a datelor stocate în registre. (Termenul de registru nu este folosit în modul cel mai riguros deoarece este vorba, de fapt, de latch-urile ce configurează memoria *RAM*).

• **Memoria RAM multiport** reprezintă structura pe care se centrează eficient o unitate de tip *RALU*. Vom considera un *RAM* cu două porturi de ieşire şi unul de intrare, conform reprezentării din figura 5.95. Într-un singur ciclu, din această memorie se pot citi două locaţii, specificate prin *LADR* şi *RADR* (adresa left şi cea right) şi se poate scrie la o a treia specificată prin *DADR* (adresa destinaţie). Schema bloc a acestei memorii este reprezentată în figura 5.96.

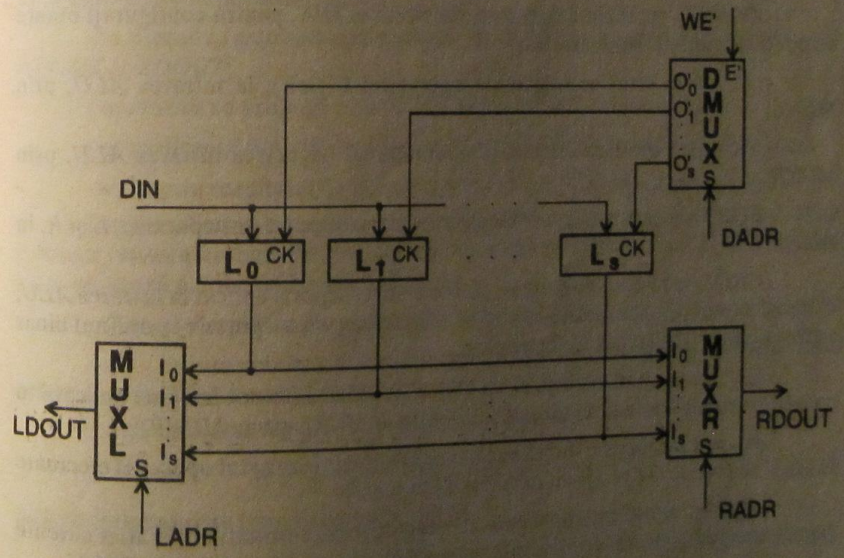



**Figura 5.95**

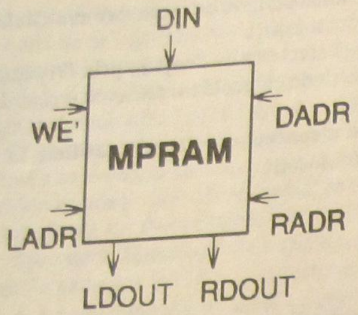



**Figura 5.96**

Aplicaţiile curente presupun că *RADR = DADR*, în sensul că scrierea se face în locaţia selectată la ieşirea *RDOUT*.

• **Structura de principiu a unui RALU** presupune un $AS^3$ cu *RAM* multiport şi *CLC'* realizat sub forma unui *ALU* la care se vor adauga multiplexoare pentru accesul datelor în sistem. O variantă curentă este cea reprezentată în figura 5.97, unde:

- *LA* - adresa operandului din stînga (*l*eft) extras din *RAM* în cîclul curent;

- *RA* - adresa operandului drept (*r*ight) şi adresa ce indică, în acelaşi timp, destinaţia rezultatului, adică latch-ul din *RAM* unde acesta va fi stocat;

- *DIN* (la intrarea *MUXL*) - reprezintă date ce sînt recepţionate din exteriorul sistemului, date ce pot deveni, eventual, operanzi la intrarea *ALU*;

- *CONST* - reprezintă o intrare, simetrică cu *DIN*, pentru configuraţi binare ce pot fi recepţionate din exterior;

- *SL* - este bitul ce selectează operandul *L* (*l*eft), la intrarea *ALU*, prin *MUXL*;

- *SR* - este bitul ce selectează operandul *R* (right), la intrarea *ALU*, prin *MUXR*;

- *FUNC* - reprezintă codul funcţiilor efectuate cu cei doi operanzi, *L* şi *R*, în *ALU*;

- *CRIN* - este bitul de *carry in*, (intrare de transport) aplicat la intrarea *ALU*; el indică faptul că operaţia aritmetică se realizează cu transport de la ordinul binar anterior sau se realizează fără transport;

- *DOUT* - reprezintă ieşirea de date a sistemului (se pot imagina aplicaţii în care să se extragă date şi de la ieşirile circuitelor *MUXL* şi/sau *MUXR*);

- *W* - este un bit care indică dacă în ciclul curent rezultatul operaţiei efectuate în *ALU* se înscrie (*W = 1*) sau nu (*W = 0*) în *RAM*;

- *IND* - sînt biţi independenţi ce caracterizează rezultatul operaţiei curente (spre exemplu *CR*); sînt consideraţi ca formînd o parte din ieşirile din *RALU*.

În prima fază a ceasului, *CK = 0*, se produc următoarele procese:



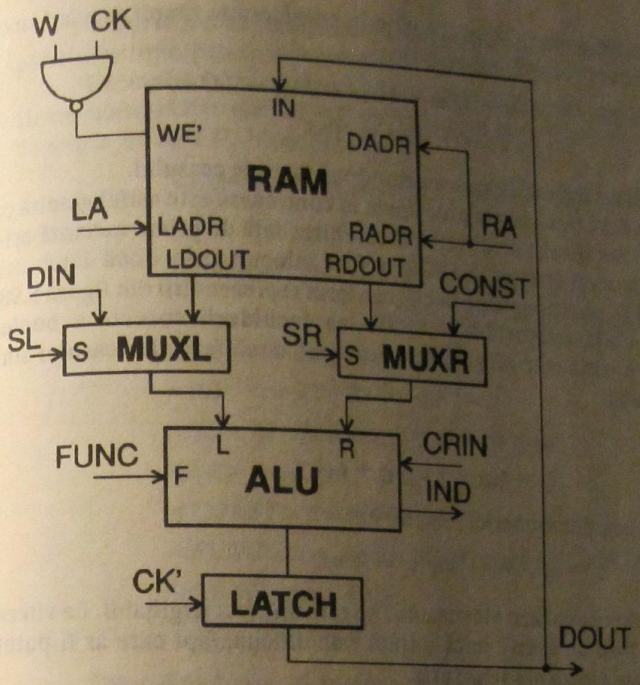



**Figura 5.97**

- ca urmare a tranziţiei negative a frontului (în exemplul nostru) în timpul $t_C$ apar comenzile la intrările *RALU*;

- în timpul $t_A$ este accesată memoria *RAM* şi sînt disponibili cei doi operanzi *LDOUT* şi *RDOUT*;

- operanzii se propagă prin *MUXL* şi/sau *MUXR* în timpul $t_{MUX}$;

- se realizează funcţia, comandată de codul *FUNC*, în *ALU* în timpul $t_{ALU}$;

- se înscrie rezultatul de la ieşirea *ALU* în *LATCH*, în timpul $t_{LATCH}$.

După realizarea acestor acţiuni ceasul poate comuta, devenind *CK = 1*. Faza a doua a ceasului trebuie folosită pentru înscrierea rezultatului în *RALU*. Timpii ce caracterizează a doua fază a ceasului sînt:

- $t_{NAND}$ - timpul de propagare prin poarta *NAND*;

- $t_W$ - durata minimă a impulsului ce comandă intrarea *WE'*, astfel încît scrierea în *RAM* să fie corect efectuată.

Fazele ceasului vor avea deci următoarele durate:

$$t_0 = t_C + t_A + t_{MUX} + t_{ALU} + t_{LATCH}$$

pentru intervalul de timp în care *CK = 0* şi

$$t_1 = t_{NAND} + t_W$$

pentru intervalul de timp cînd $CK = 1$. Luînd în consideraţie timpii asociaţi unei tehnologii *TTL* standard rezultă, aproximativ:

$$t_0 = 35ns + 35ns + 15ns + 50ns + 20ns = 155ns,$$

$$t_1 = 15ns + 25ns = 40ns.$$

Observăm o foarte mare dezechilibrare a celor două faze ale ceasului.

• **Varianta de RALU** pe care o propunem în continuare este astfel gîndită ca valorile $t_0$ şi $t_1$ să nu mai fie atît de diferite. Deosebirea faţă de prima variantă este dată de faptul că latch-ul master *LATCH* a fost înlocuit prin două latch-uri, *LATCH L* şi *LATCH R*, poziţionate pe buclă, conform reprezentării din figura 5.98. Efectul lor este acelaşi, asigurînd posibilitatea închiderii corecte a buclei automatului, numai poziţia este alta. În acest caz cele două faze ale ceasului sînt date după cum urmează:

$$t_0 = t_C + t_A + t_{LATCH},$$

$$t_1 = t_{MUX} + t_{ALU} + t_W.$$

Considerînd acelaşi exemplu numeric, rezultă:

$$t_0 = 90ns \text{ iar } t_1 = 90ns.$$

Echilibrarea celor două faze ale ceasului nu este un efect neglijabil. La viteze mari, obţinerea unei faze de foarte mică durată este dificilă, fapt care ar fi putut limita performanţele de viteză ale structurii.

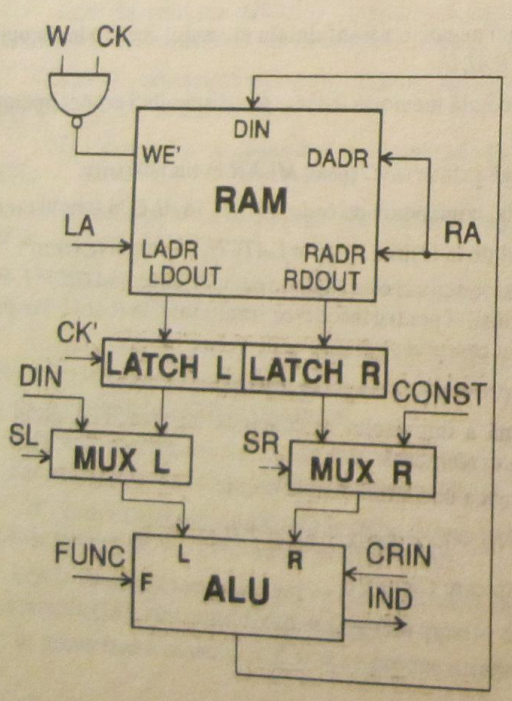



**Figura 5.98**



• **Specificarea funcţională a unei structuri** RALU o vom face, într-un caz particular tipic, printr-un exemplu.

**Exemplul 5.33** În paragraful 5.6.2 atunci cînd defineam *<Linie de comandă>* lăsam neexplicitat cîmpul *<Com>*. Parţial acest cîmp poate fi folosit pentru a comanda un *RALU*. Vom putea scrie:

$$<Com> ::= <Com\ RALU><Alte\ Com>$$

unde

$$<ComRALU> ::= <LA><RA><SL><SR><CONST><FUNC><CRIN><WRITE>$$

$$<LA> ::= L0 \mid L1 \mid \ldots LF \mid -,$$

$$<RA> ::= R0 \mid R1 \mid \ldots RF \mid -,$$

$$<SL> ::= DIN \mid -,$$

$$<SR> ::= K \mid -,$$

$$<CONST> ::= \#0000 \mid \#0001 \mid \ldots \mid \#FFFF \mid -,$$

$$<FUNC> ::= AND \mid OR \mid XOR \mid PLUS \mid MIN \mid LEFT \mid RIGHT \mid SHR \mid -,$$

$$<CRIN> ::= CRY \mid -,$$

$$<WRITE> ::= W \mid -,$$

Deci *RALU* are 16 registre, de la *0* la *F*, numerotate în hexazecimal. Dacă *DIN* şi/sau *K* nu apar în linia de comandă, atunci operanzii sînt cei de la ieşirile *RAM*-ului, selectaţi prin codurile *LA* şi/sau *RA*. Registrele interne (locaţiile din *RAM*) sînt de 16 biţi, impuşi de constanta maximă folosită ($FFFF_H$). *ALU* are funcţii logice (primele trei), aritmetice (următoarele două), de transfer (*LEFT* - trece operandul *L* şi *RIGHT* - trece operandul *R*) şi o funcţie de deplasare, la dreapta (*SHR* - *sh*ift *r*ight).

În această etapă vom putea completa parţial şi cîmpul *<Test>* ce nu a fost definit în paragraful 5.6.2.

$$<Test> ::= CR \mid OVFL \mid ZERO \mid EQUAL \mid PAR \mid SGN \mid \ldots.$$

Acest cîmp cuprinde de fapt, pe lîngă alţi posibili indicatori, indicatorii, *IND*, de la ieşirea *ALU* din *RALU*, unde:

- *CR* - ieşirea de depăşire (*carry* pentru cazul în care se efectuează funcţia *PLUS* sau *borrow* cînd funcţia comandată este *MIN* asupra întregilor pozitivi);
- *OVFL* - indică depăşirea capacităţii de reprezentare în cazul numerelor cu semn;
- *ZERO* - indică faptul că rezultatul curent are valoarea zero;
- *EQUAL* - indică egalitatea celor doi operanzi la intrarea *ALU*;
- *PAR* - este bitul cel mai puţin semnificativ al rezultatului (indică paritatea numărului asociat reprezentării binare);
- *SGN* - este bitul cel mai semnificativ al rezultatului, ce poate fi folosit pentru a marca semnul atunci cînd configuraţia binară este interpretată drept un număr. □

Observăm că structura *<Com RALU>* cuprinde codurile de comandă ale *RALU* anterior explicate. Aceste coduri pot fi generate de către o unitate de control de tip *CROM*, spre exemplu.

**Exemplul 5.34** O posibilă secvenţă de comenzi către *RALU* ar putea fi următoarea:

$00_H$ *DIN LEFT RO W*
$01_H$ *K #A125 R1 RIGHT W*
$02_H$ *L0 R2 LEFT W*
$03_H$ *L1 R0 PLUS W*
$04_H$ *L1 R1 XOR W*

Efectul comenzii de pe linia $00_H$ este următorul: configuraţia binară de 16 biţi de la intrarea *DIN* este înscrisă în locaţia *0* a memoriei *RAM*. Următoarea comandă încarcă constanta $A125_H$ la locaţia de memorie *1*. În următorul pas conţinutul registrului *0* este transferat în registrul *2*, apoi, numerele stocate în registrele *0* şi *1* sînt adunate modulo $10000_H$ şi rezultatul este depus la adresa *0*. Ultima comandă determină aducerea la zero a locaţiei *1* din *RAM*. Observăm că nu se creează nici o confuzie dacă în linia $01_H$, *mnemonica K nu este specificată*, deoarece lucrul cu o constantă este indicat prin explicitarea acesteia. Programul care generează codurile binare asociate fiecarei linii poate genera corect codul din cîmpul *<SR>* drept urmare a explicitării constantei. □

• Unităţile de tip *RALU* sînt gîndite ca cipuri ce pot fi astfel conectate încît să se *extindă* dimensiunea cuvîntului prelucrat. De regulă, se poate încapsula într-un cip un *RALU* de patru biţi, definindu-se o serie de semnale care permit interconectarea a *n* cipuri pentru realizarea unui *RALU* de *4n* biţi. (Circuitul care s-a impus cel mai puternic este circuitul *bit-slice AMD2901*, un *RALU* de 4 biţi deosebit de performant. Structura şi funcţiile lui pot fi găsite în cataloagele firmelor producătoare.)

## 5.8 Automate orientate funcţional

Automatele finite, la fel ca şi circuitele combinaţionale, pot fi astfel gîndite ca să corespundă unor funcţii bine definite. Circuitelor combinaţionale specializate (subcapitolul 3.5) le corespund automatele orientate funcţional. Exemplele 5.11 şi 5.12 sînt elocvente în acest sens. Primul este un *acumulator modulo* $2^n$ care poate realiza sumarea unui şir de numere recepţionate pe intrarea *X* (figura 5.43), starea automatului fiind dată de suma parţială anterior realizată. Deci, automatul este orientat către *funcţia de acumulare*. Al doilea exemplu se constituie într-un circuit ce pune în evidenţă, în ordine, biţii egali cu *1* ai unei configuraţii binare de *n* biţi. Circuitul inspectează rapid o configuraţie binară indicîndu-i compoziţia, fără a face



nici un pas inutil, în sensul că în fiecare ciclu de ceas pune în evidenţă poziţia unui bit egal cu *1*. Ca şi circuitul anterior, acesta este configurat cu circuite standard (figura 5.44) din categoria circuitelor combinaţionale specializate.

Pot fi imaginate şi alte circuite, din categoria sistemelor de ordinul doi, ce au o funcţie generală, utilizabilă în mai multe tipuri de aplicaţii. Proiectarea unor astfel de structuri nu se realizează cu procedeele standard anterior expuse ci presupune, în majoritatea cazurilor, o definire în limbaj natural. Numărul foarte mare de stări pe care-l presupun aceste automate face chiar imposibilă aplicarea procedeelor formale standard.

### 5.8.1 Acumulator de produse

O clasă largă de aplicaţii numerice presupune evaluarea unor expresii de forma:

$$E = a_1 b_1 + a_2 b_2 + \ldots + a_i b_i + \ldots + a_n b_n$$

unde $a_i$ şi $b_i$ sînt reprezentate binar cu $p$ biţi. Pentru $n$ finit, limitat la valoarea $2^q$ se poate folosi automatul din figura 5.99, al cărui circuit combinaţional cuprinde un înmulţitor şi un sumator astfel încît, dacă

$$CLR = 0\text{, atunci } Q^+ = (Q\ plus(a_i \times b_i))_{mod\ 2^{2p+q}}$$

iar dacă

$$CLR = 1\text{, atunci } Q^+ = zero.$$

Mărimile $a_i$ şi $b_i$ sînt sincrone cu ceasul, *CK*, al automatului.

Pentru acest automat $Q \equiv Y$, iar în spaţiul stărilor există un număr de $2^{2p+q}$ stări distincte.

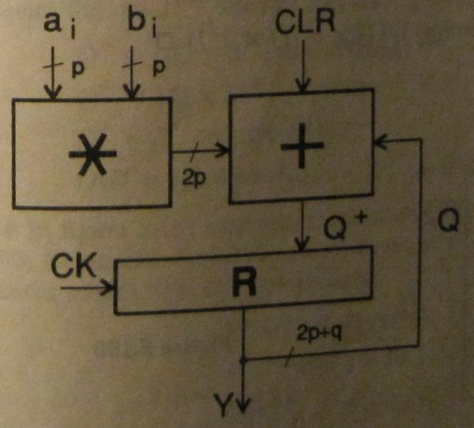



**Figura 5.99**

### 5.8.2 Memorie de tip stivă (*LIFO*)

O memorie de tip stivă permite stocarea unui şir de simboluri ce poate fi accesat numai la unul din capete, atît pentru scriere, cît şi pentru citire. Modul de funcţionare este sugerat şi prin denumirea pe care o are în limba engleză: *LIFO* (*last in first out*). Deci, *ultima* înregistrare este şi *prima* citită.

Dacă în memorie este înregistrat şirul *abca*, prin scrierea simbolului *c* conţinutul memoriei devine *abcac*. Dacă se fac două citiri succesive şirul stocat devine pe rînd *abca* şi *abc*. Accesul la simbolurile din şir este realizat, evident, la extremitatea dreaptă a acestuia. Citirea simbolurilor din memorie se face în ordinea inversă scrierii. Curent, comanda de scriere este numită *PUSH*, pentru că şirul este *împins* înspre stînga, mai adînc în memorie, pentru a se face loc noului simbol. Pentru citire se foloseşte denumirea *POP* prin care se extrage simbolul din vîrful stivei.

Setul de comenzi dat unei memorii *LIFO* va fi: *PUSH, POP, NOOP*. Ultima funcţie este importantă pentru că vor exista intervale de timp în care conţinutul memoriei se va conserva, chiar dacă ceasul sistemului va funcţiona. Schema de principiu a unei astfel de memorii este reprezentată în figura 5.100, unde pe *C* se aplică una din cele trei comenzi anterior listate iar pe *IN* şi *OUT* se realizează transferul de simboluri memorate.

O astfel de memorie este un *automat finit*. Pentru cazul unor cuvinte de un bit descrierea formală a memoriei este dată de următoarea definiţie.

**Definiţia 5.16** O memorie *LIFO* pentru $n$ cuvinte de un bit este un automat finit a cărui stare $Q$ comută după următoarele reguli:

$$Q^+ = ((Q \times 2^1)\ plus\ IN)_{mod 2^n},\ \text{pentru } PUSH,$$
$$Q^+ = \lfloor Q \times 2^{-1} \rfloor,\ \text{pentru } POP,$$
$$Q^+ = Q,\ \text{pentru } NOOP,$$

unde $Q$ reprezintă, sub forma unui număr binar, starea curentă a automatului, $Q^+$ reprezintă starea următoare, prin $\lfloor N \rfloor$ notăm partea întreagă din numărul $N$, iar $n$ este numărul maxim de biţi ce pot fi stocaţi în *LIFO*;

$$OUT = (Q\ minus\ ((\lfloor Q \times 2^{-1} \rfloor) \times 2^1)).\ \square$$

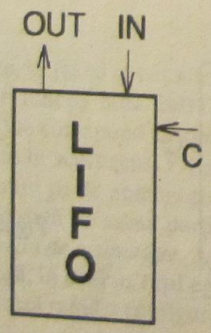



**Figura 5.100**



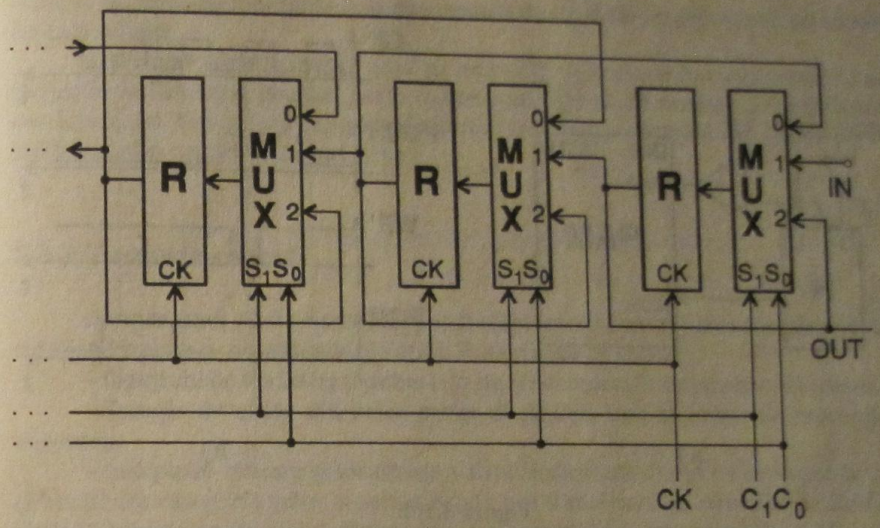



**Figura 5.101**

**Într-adevăr, la scriere (*PUSH*) conţinutul memoriei se deplasează cu o poziţie la stînga (numărul $Q$, ce reprezintă starea, se înmulţeşte cu doi) şi se adaugă bitul de pe intrarea *IN*. La extragere simbolul din vîrful stivei este extras (*POP*), deci configuraţia binară curentă, $Q$, este deplasată cu o poziţie la dreapta, operaţie realizată prin împărţirea întreagă cu doi. Simbolul de la ieşire este bitul cel mai puţin semnificativ al numărului $Q$ obţinut prin scăderea din acesta a configuraţiei binare rezultate prin ştergerea din $Q$ a bitului cel mai puţin semnificativ.**

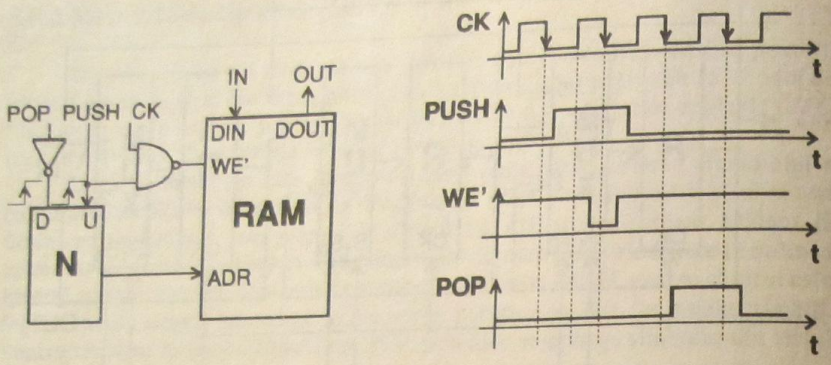
Cazul general presupune memorarea unor cuvinte de $m$ biţi.

**Definiţia 5.17** O memorie *LIFO* pentru $n$ cuvinte de cîte $m$ biţi fiecare este un automat finit cu $2^{n \times m}$ stări a cărui stare $Q$ evoluează după cum urmează:

$$Q^+ = ((Q \times 2^m)\ plus\ IN)_{mod 2^{n \times m}},\ \text{pentru } PUSH,$$
$$Q^+ = \lfloor Q \times 2^{-m} \rfloor,\ \text{pentru } POP,$$
$$Q^+ = Q,\ \text{pentru } NOOP,$$
$$OUT = (Q\ minus\ ((\lfloor Q \times 2^{-m} \rfloor) \times 2^m)).\ \square$$

• În figura 5.101 este reprezentată schema de principiu a unei memorii de tip *LIFO*. Remarcăm faptul că este vorba de un registru de deplasare bidirecţional. Codificarea comenzilor este următoarea:

$$PUSH:\ C_1C_0 = 01,$$
$$POP:\ C_1C_0 = 00,$$
$$NOOP:\ C_1C_0 = 10.$$




a) b)

**Figura 5.102**

(Codul $C_1C_0 = 11$ ar putea fi folosit pentru a comanda resetarea întregului continut al stivei.) Structura prezentată pune în evidenţă, odată în plus, ordinul doi pe care îl are sistemul.

• O variantă posibilă, realizată tot ca sistem de ordinul doi, presupune utilizarea unui numărător reversibil şi a unei memorii *RAM*, conform reprezentării din figura 5.102*a*. Şirul de simboluri este stocat la locaţii succesive în memoria *RAM*. Vîrful stivei este adresat prin starea numărătorului *N*. Dacă în prima variantă elementele şirului se deplasau în registrul bidirecţional, în aceasta ele rămîn stocate la aceleaşi locaţii, deplasîndu-se numai pointerul dat de numărul ce reprezintă starea numărătorului. La fiecare *PUSH* starea numărătorului se incrementează iar apoi simbolul de pe intrarea *IN* se scrie în memorie la comanda $WE' = 0$, conform diagramelor de timp din figura 5.102*b*. Din aceleaşi diagrame rezultă că pentru comanda *POP* vîrful stivei poate fi preluat de la ieşirea *OUT*, iar cu frontul posterior al comenzii se comandă decrementarea numărătorului, adresîndu-se astfel noul vîrf al stivei.

• Avantajul celei de a doua variante este evident pentru capacităţi mari de memorare, deoarece pentru fiecare bit "stivuit" se foloseşte numai un singur latch, faţă de cazul variantei cu registru bidirecţional de deplasare în care pentru fiecare bit este necesar un bistabil de tip master-slave, deci sînt folosite două latch-uri. La capacităţi mici de memorare, de ordinul a cîteva cuvinte, simplitatea structurală impune totuşi prima soluţie.

• Ordinul structural minim asociat funcţiei de memorie *LIFO* este doi. Această funcţie de memorare nu poate fi realizată cu un sistem de ordin inferior lui doi, deoarece presupune tranziţii ce ţin cont şi de întreaga stare în care se află sistemul. Cu o memorie de tip stivă operăm asupra unui şir de simboluri ce este privit în fiecare etapă ca un *tot*. Şi într-un *RAM* poate fi stocat un şir de simboluri dar orice modificare nu se realizează decît printr-o acţiune *locală* ce nu afectează



*întregul* şir. Din acest motiv funcţia de memorie de tip *RAM* nu presupune un sistem de ordinul doi.

Prelucrări *globale* asupra unui şir nu pot fi realizate decît cu structuri ce au cel puţin ordinul doi, deci cel puţin cu structuri care sînt automate. Dimensiunea configuraţiei binare ce reprezintă starea este proporţională cu numărul de simboluri din care este format şirul.

### 5.8.3 Memoria conexă

Asupra unui şir de simboluri por fi efectuate şi operaţii mai complexe decît cele realizate cu o memorie de tip stivă. Putem avea în vedere:

- funcţiuni de tip *insert* şi *delete* într-un şir oarecare în orice punct al acestuia;
- funcţia de citire din orice punct al şirului, stabilit printr-un mecanism oarecare;
- funcţia de căutare şi localizare a unui simbol sau a unui şir de simboluri.

(Funcţii din categoria celor anterior listate pot fi realizate pe structuri de date de tip listă în sisteme de calcul. Reprezentarea acestor liste este o mare şi ineficientă consumatoare de memorie. Se asigură o reprezentare fizic distribuită într-un spaţiu de memorie astfel încît simboluri alăturate, conexe, în listă să poată exista de fapt localizate în zone net distincte în memorie. De aici o serie de dificultăţi în utilizarea acestor reprezentări.)

Vom descrie în continuare un automat finit cu spaţiul stărilor structurat care realizează funcţii utile de operare pe şiruri de simboluri. Îl vom denumi *memorie conexă* pentru că asigură o reprezentare a şirurilor care, indiferent de prelucrările la care sînt supuse, îşi păstrează adiacenţa fizică a simbolurilor ce formal sînt adiacente.

**Definiţia 5.18** Memoria conexă, notată *MC*, este un automat finit, cu spaţiul stărilor structurat, de forma

$$MC = (X \times C, Y, Q_0 \times Q_1 \times \ldots \times Q_n, \lambda, \delta)$$

unde

$$X \equiv Y \equiv S, \quad Q_i \equiv S \cup S^* \text{ pentru } i = 0,1,\ldots,n,$$
$$S = \{s_0, s_1, \ldots, s_m\},$$
$$S^* = \{s_0^*, s_1^*, \ldots, s_m^*\},$$

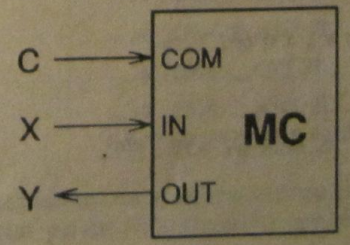



**Figura 5.103**

iar $C$ este mulţimea comenzilor

$$C = \{UF\,s,\ CF\,s,\ INS\,s,\ RST\,s,\ DEL,\ NOP,\ RD,\ BAC\}.$$

Funcţiile de tranziţie ale stării şi ieşirii vor fi date odată cu explicitarea elementelor din $C$. Pentru aceasta convenim asupra următoarelor notaţii:

- $(q_0, q_1, ..., q_n)$ este configuraţia instantanee a memoriei, $q_i$ fiind variabile cu valori în $S \cup S^*$;

- $q_i = s$ semnifică faptul că elementul $q_i$ al şirului are valoarea $s$;

- $q_i = s^*$ semnifică faptul că elementul $q_i$ al şirului are valoarea $s$ şi este marcat;

- $p = min\ (i \mid val\ (q_i) \in S^*)$, $p$ este indicele primului simbol marcat din şir;

- $s \in S$ iar $s^* \in S^*$ sînt cele două forme ale simbolurilor ce formează şirurile stocate în $MC$ (spunem că $s^*$ se obţine prin *marcarea* simbolului $s$);

- $q_i \leftarrow s$ semnifică atribuirea valorii $s$ variabilei $q_i$ (simbolul de pe poziţia $i$ din şirul stocat în $MC$);

- $q_i \leftarrow s^*$ semnifică atribuirea valorii $s^*$ variabilei $q_i$ (adică, $q_i$ are valoarea $s$ şi este marcat);

- $q_i \leftarrow q_i^*$, valoarea variabilei $q_i$ este marcată;

- $q_i^* \leftarrow q_i$, valoarea variabilei $q_i$ este demarcată;

- $q_i \leftarrow q_j$, variabila $q_i$ ia valoarea variabilei $q_j$.

Explicităm în continuare elementele din $C$.

- *UF s* : căutare necondiţionată (*u*nconditional *f*ind)

  $q_{i+1} \leftarrow q_{i+1}^*$ pentru $\forall\, q_i = s,\ s \in S$, toate celelalte simboluri din şir fiind demarcate;

- *CF s* : cătare condiţionată (condiţional *f*ind)

  pentru $\forall\, q_i = s^*$, unde $s^* \in S^*$, $q_i^* \leftarrow q_i$ iar $q_{i+1} \leftarrow q_{i+1}^*$, toate celelalte simboluri fiind demarcate;

- *INS s* : *ins*erarea simbolului $s$

  $q_{i+1} \leftarrow q_i$ pentru $p \le i < n$, iar $q_p \leftarrow s$;

- *RST s* : *res*etarea la valoarea $s$

  $q_i \leftarrow s$ pentru $\forall\, i,\ i = 0, 1, ..., n$;

- *DEL* : *del*ete

  $q_i \leftarrow q_{i+1}$ pentru $p \le i < n$, $q_n \leftarrow q_n$,
  $OUT = q_p$;

- *NOP* : *no op*eration

  $OUT = q_p$;

- *RD* : *read*

  $q_p^* \leftarrow q_p$, $q_{p+1} \leftarrow q_{p+1}^*$,
  $OUT = q_p$;

- *BAC* : *back*ward

  $q_{p-1} \leftarrow q_{p-1}^*$, $q_p^* \leftarrow q_p$,
  $OUT = q_p$. □

**Exemplul 5.35** Vom considera că şirul stocat în *MC* este, într-o anumită porţiune, de forma

... *A ANA 9 ANI B* ...

şi dorim să-l modificăm astfel încît să marcăm faptul că *ANA* are *10* ani. Vom genera următoarea secvenţă de comenzi (notăm cu ◊ blanc-ul):

*UF* ◊
*CF A*
*CF N*
*CF A*
*CF* ◊
**while** *OUT* = ◊
*DEL*
**repeat**
*INS 1*
*INS 0.*

Indicînd prin subliniere simbolurile marcate, evoluţia şirului, drept urmare a secvenţei de comandă anterior listată, este următoarea:

... *A ANA 9 ANI B* ... (*UF* ◊)
... *A ANA 9 ANI B* ... (*CF A*)
... *A ANA 9 ANI B* ... (*CF N*)
... *A ANA 9 ANI B* ... (*CF A*)
... *A ANA 9 ANI B* ... (*CF* ◊)
... *A ANA ANI B* ... (*DEL*)
... *A ANA 1 ANI B* ... (*INS 1*)
... *A ANA 10 ANI B* ... (*INS 0*). □

• **Automatul cu memorie conexă** (*AMC*) este reprezentat în figura 5.104, interconectat pe un *BUS* standard prin intermediul unei interfeţe. *AMC* poate

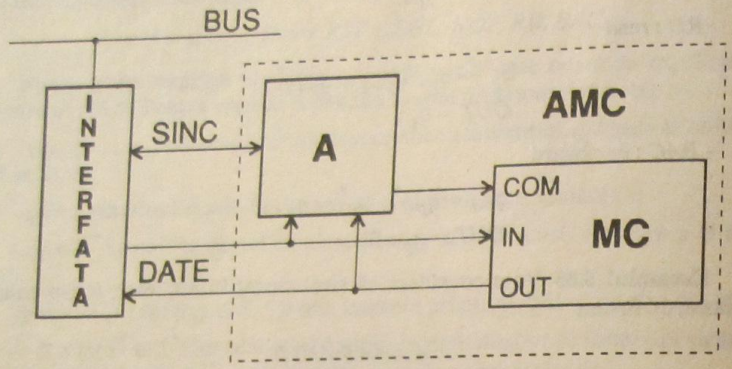



**Figura 5.104**

primi şiruri de simboluri prin interfaţă sau poate emite din *MC* şiruri de simboluri în exterior. De asemenea, *AMC* poate prelucra şirurile de simboluri prin secvenţe de comenzi generate de automatul *A* către *MC*.

Funcţionarea *AMC* poate fi definită printr-un ciclu ce cuprinde următoarele etape:

*READ (şir de simboluri)*

*EVAL*

*PRINT (şir de simboluri).*

În prima etapă este recepţionat un şir de simboluri ce este, eventual, inserat în *MC*. A doua etapă presupune o acţiune oarecare a automatului *A* asupra conţinutului global al *MC*. În ultima etapă un subşir din *MC* poate fi emis drept rezultat. *Exemplul 5.35* poate reprezenta o procedură de *EVAL* aplicată prin *A* conţinutului memoriei *MC*.

## 5.9 Complexitatea combinaţională a automatelor finite

Avem nevoie, oare, de automate? Nu am putea rezolva problemele în care implicăm aceste structuri prin circuite mai simple, prin cele mai simple, eventual: circuitele logice combinaţionale?

De regulă, la intrarea unui automat finit se aplică un şir finit de simboluri, să-l notăm cu $x_1, x_2, ..., x_n$, şi la ieşire se obţine tot un şir finit de configuraţii binare, notat cu $y_1, y_2, ..., y_n$. În tactul *1* se aplică $x_1$, în tactul doi $x_2$ ş.a.m.d. Indicele va fi



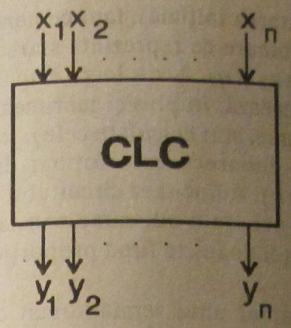



**Figura 5.105**

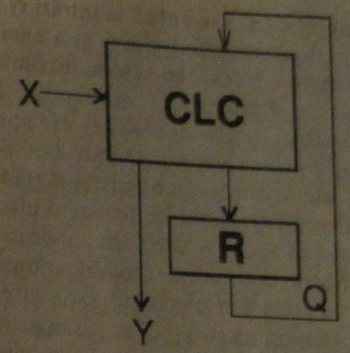



**Figura 5.106**

deci asociat tactului în care un element din $X$ este aplicat automatului. Deci $i$ din $x_i$ nu spune nimic despre valoarea aplicată intrării ci numai despre momentul în care o valoare din $X$ este aplicată automatului. Similar, $y_i$ reprezintă simbolul de pe poziţia $i$ în şirul celor emise la ieşirea automatului. Deci, funcţionarea timp de $n$ tacturi a unui automat finit poate fi simulată printr-un *CLC* de tipul celui din figura 5.105, unde la intrare sînt aplicate cele $n$ simboluri ale şirului de intrare, iar la ieşire se obţin cele $n$ simboluri ale şirului de ieşire. Cît de complex este acest circuit combinaţional? Cum se corelează, eventual, complexitatea cu adîncimea lui?

Dacă ar fi să aplicăm teoria generală, fără a particulariza pentru modul în care un *CLC* s-ar configura pentru un automat finit, ar rezulta că $S(n)$ ar fi de tip exponenţial pentru adîncime constantă. Dar, pentru cazul particular în studiu şi pentru adîncimi ce nu se doresc constante, s-ar putea să obţinem rezultate interesante, diferite de acesta.

### 5.9.1 Conversia automat finit - circuit combinaţional

Să considerăm automatul, iniţial în $q_0$, reprezentat în figura 5.106, căruia i se aplică la intrare secvenţa de $n$ simboluri, $x_1, ..., x_n$, la care răspunde prin $y_1, ..., y_n$, trecînd prin stările $q_1, ..., q_n$. (La fel ca pentru $x_i$ şi $y_i$, $q_i$ reprezintă starea automatului după aplicarea a $i$ impulsuri de ceas. Automatul nu are $n$ stări, ci comută sub influenţa a $n$ tacturi.) Acest automat rezolvă problema prelucrării şirului $x_1, ..., x_n$ cu un circuit a cărui complexitate este $S_1(n) = O(1)$ într-un timp $T_1(n) = O(n)$. Într-adevăr dimensiunea *CLC*-ului asociat automatului finit nu depinde de lungimea şirului aplicat la intrare, este constantă. Timpul de prelucrare este dat de cele $n$ tacturi care comută automatul din $q_0$ pînă în $q_n$, permiţînd generarea şirului $y_1, ..., y_n$.

În figura 5.107 este reprezentat circuitul combinaţional care rezolvă aceeaşi problemă: dîndu-se şirul $x_1, ..., x_n$, circuitul generează şirul $y_1, ..., y_n$. Circuitul este format din *CLC*-uri identice între ele şi cu *CLC*-ul automatului din figura 5.106.

Primului *CLC* i se aplică la intrări $x_1$ şi starea $q_0$ (starea iniţială), fapt ce permite calculul mărimii de ieşire $y_1$ şi a unei configuraţii binare ce reprezintă starea $q_1$. Ieşirea $y_2$ se calcula cu automatul finit pornind de la $x_2$ şi $q_1$. Acest lucru este făcut de cel de al doilea *CLC* din figura 5.107, care generează, în plus şi mărimea $q_2$, necesară în continuare. Astfel, din aproape în aproape, sînt calculate cele $n$ ieşiri. Complexitatea acestui circuit este $S_2(n) = O(n)$, deoarece este format din $n$ *CLC*-uri de dimensiune constantă (independentă de $n$). Adîncimea circuitului va fi $D_2(n) = O(n)$, deoarece fiecare dintre cele $n$ *CLC*-uri are o adîncime constantă. Rezultă că $T_2(n) = O(n)$, timpul de calcul al secvenţei de ieşire fiind proporţional cu adîncimea circuitului combinaţional.

Putem lua în consideraţie şi cazul particular al unui semiautomat care, recepţionînd un şir $x_1,..., x_n$, ajunge în starea finală $q_n$, pornind din $q_0$. Sirul va fi caracterizat prin starea finală în care ajunge automatul. În acest caz se pune problema de a simula un *semiautomat* prin circuitul combinaţional din figura 5.108.

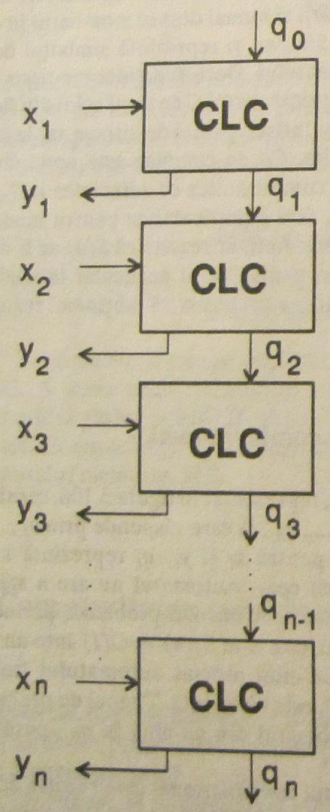



Figura 5.107

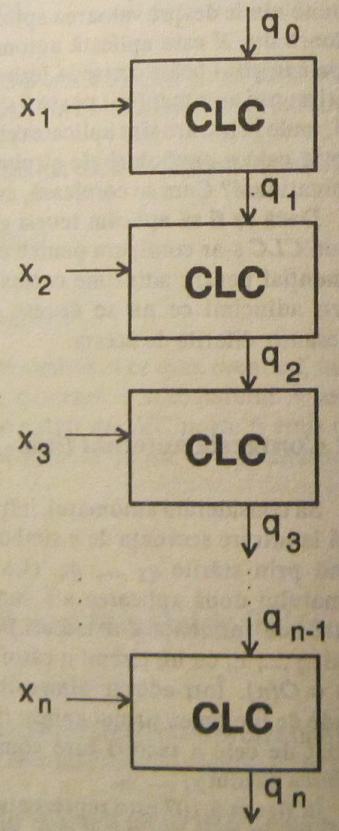



Figura 5.108



Diferenţa nu este esenţială faţă de problema conversiei unui automat. În continuare vom discuta, fără să afectăm esenţial rezultatele, despre semiautomate.

Se pune întrebarea: putem reconfigura echivalentul combinaţional al unui semiautomat într-un circuit cu o adîncime mai mică de $D(n) = O(n)$ fără să creştem complexitatea inacceptabil de mult? De asemenea sîntem interesaţi ca această reconfigurare să fie obţinută ca o regulă generală, aplicabilă oricărui semiautomat.

### 5.9.2 Echivalentul combinaţional al unui semiautomat, cu $D(n) = O(\log n)$

Conform teoremei lui Spira o structură combinaţională de tipul celei din figura 5.108 se poate transforma într-o *formulă* sau un *circuit* (conform definiţiilor din subcapitolul 3.6) de adîncime $D(n) = O(\log n)$. Pentru a face o expunere mai comodă şi mai clară ne vom servi de un exemplu.

**Exemplul 5.36** Considerăm un semiautomat finit cu patru stări, $q_0,..., q_3$, ce primeşte la intrare secvenţe formate din opt simboluri: $x_1x_2...x_8$. Starea finală în care ajunge semiautomatul poate fi determinată cu ajutorul unui circuit realizat prin înlănţuirea a opt *CLC*-uri identice cu cel din bucla semiautomatului, conform paragrafului anterior (figura 5.108). Pentru a realiza o reprezentare mai simplă vom folosi notaţia din figura 5.109*a* pentru un *CLC* căruia i se aplică la intrare simbolul $x_i$ şi starea $q_j$. Rezultă circuitul din figura 5.110*a* (starea iniţială a automatului este $q_0$).

Aplicînd teorema lui Spira rezultă, într-o primă etapă, structura combinaţională din figura 5.110*b*. Am folosit reprezentarea simplificată, sugerată

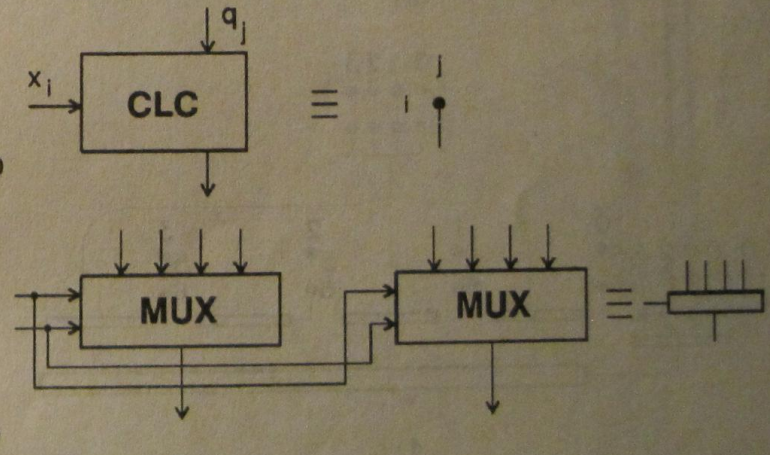
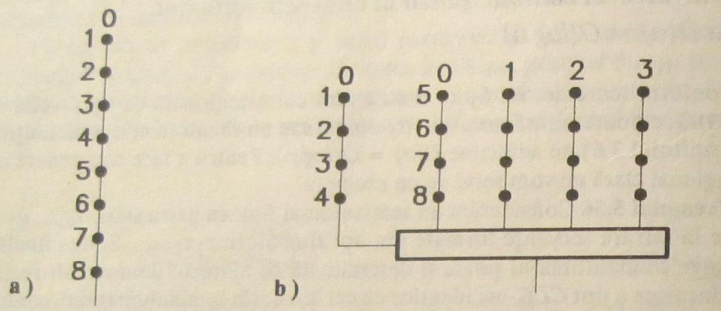
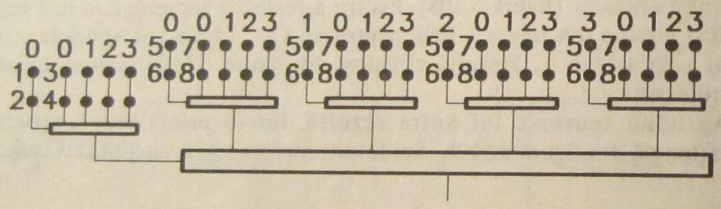



Figura 5.109

de figura 5.109*b*, pentru cele două *MUX*-uri de patru căi presupuse de aplicarea teoremei. La o a doua aplicare a procedeului rezultă configuraţia de tip formulă din figura 5.110*c*. Observăm că *CLC*-urile ce au aplicate la intrare simbolurile $x_7$ şi $x_8$ formează patru grupuri identice. Se poate trece la o configuraţie de tip circuit acceptînd ca ieşirile acestor subreţele să fie caracterizate prin *fan-out* = 4. Obţinem configuraţia de circuit din figura 5.110*d*, care este mai simplă şi implică o expandare mai puţin explozivă.



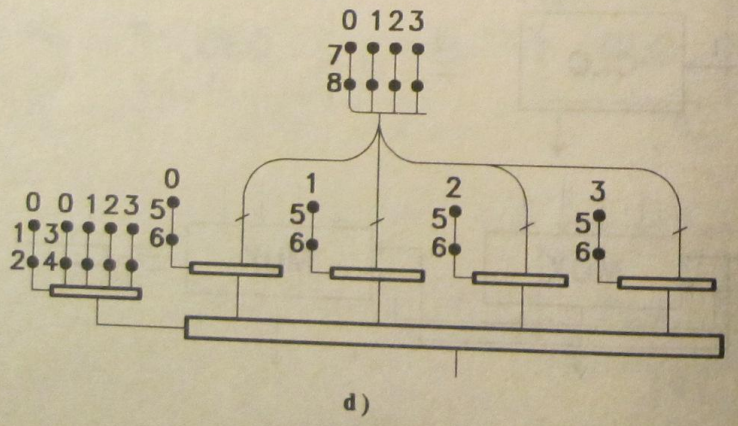




**Figura 5.110**



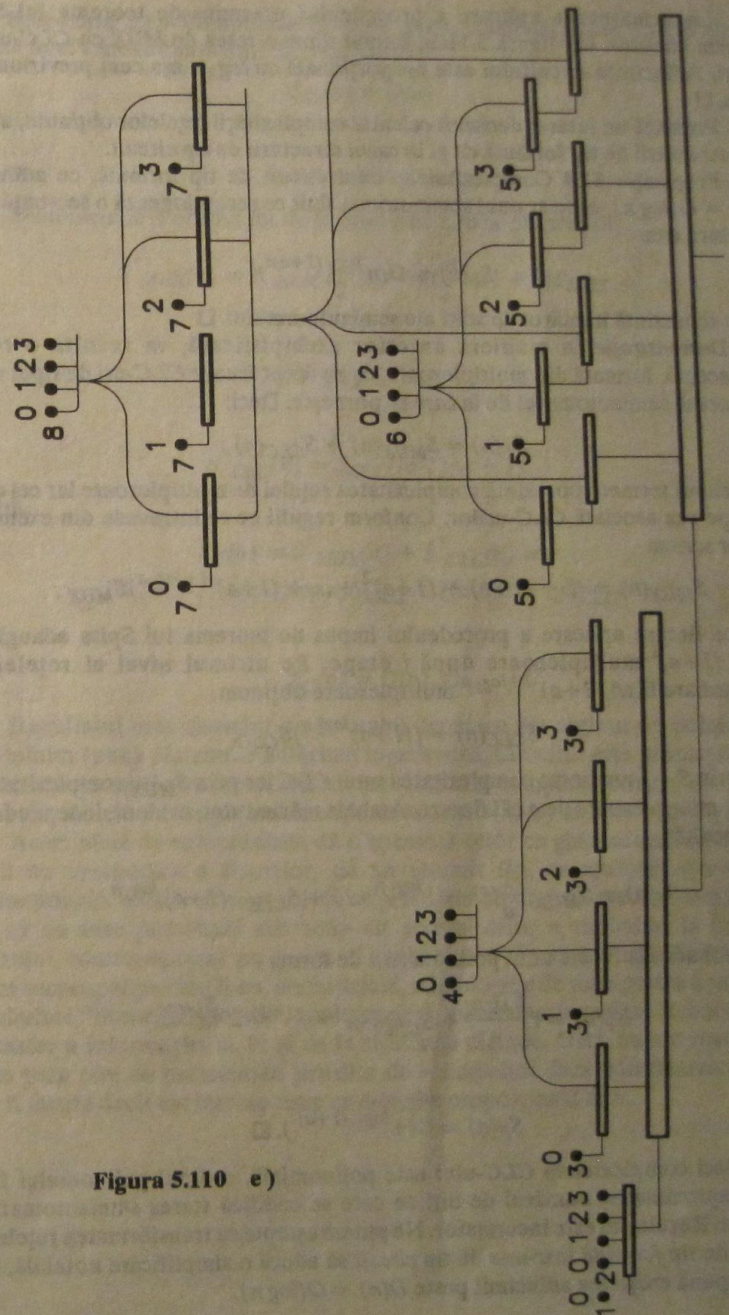



**Figura 5.110 e)**

La următoarea aplicare a procedeului presupus de teorema lui Spira obţinem circuitul din figura 5.110*e*, format dintr-o reţea de *MUX* cu *CLC*-uri pe intrări. Adîncimea circuitului este proporţională cu $log_2 n$ aşa cum previzionează teoria. □

Prezintă un interes deosebit calculul complexităţii reţelelor obţinute, atît în cazul structurii de tip formulă cît şi în cazul structurii de tip circuit.

**Propoziţia 5.14** Complexitatea unui circuit de tip formulă, cu adîncime $D_F(n) = O(\log n)$, asociat unui semiautomat finit ce recepţionează o secvenţă de $n$ simboluri, este

$$S_F(n) = O(n^{\log_2(1+a)})$$

unde $a$ reprezintă numărul de stări ale semiautomatului. □

*Demonstraţie* În maniera anterior exemplificată, va rezulta o reţea arborescentă, formată din multiplexoare, ce au drept frunze *CLC*-uri de tipul celui ce formează semiautomatul de la care se porneşte. Deci:

$$S_F(n) = S_{MUX}(n) + S_{CLC}(n)$$

unde primul termen reprezintă complexitatea reţelei de multiplexoare iar cel de al doilea pe cea asociată *CLC*-urilor. Conform regulii ce se întrevede din exemplul anterior scriem

$$S_{MUX}(n) = (1 + (1+a) + (1+a)^2 + \ldots + (1+a)^{-1+\log_2 a})S_{MUX},$$

deoarece fiecare aplicare a procedeului impus de teorema lui Spira adaugă un număr $(1+a)^i$ multiplexoare după $i$ etape. Pe ultimul nivel al reţelei de multiplexoare fiind $(1+a)^{-1+\log_2 a}$ multiplexoare obţinem

$$S_{CLC}(n) = ((1+a)^{\log_2 a})S_{CLC}.$$

Prin $S_{CLC}$ am notat complexitatea unui *CLC* iar prin $S_{MUX}$ complexitatea a $\lceil \log_2 a \rceil$ multiplexoare de $a$ căi fiecare. Ambele mărimi sînt, evident, independente de $n$. Rezultă

$$S_F(n) = S_{MUX} \cdot \frac{1}{a}((1+a)^{\log_2 n} - 1) + S_{CLC} \cdot (1+a)^{\log_2 n}$$

ce poate fi adus la forma unui polinom în $n$ de forma

$$S_F(n) = (\frac{S_{MUX}}{a} + S_{CLC}) \cdot n^{\log_2(1+a)} - \frac{S_{MUX}}{a}.$$

Deci:

$$S_F(n) = O(n^{\log_2(1+a)}). \square$$

Deci complexitatea *CLC*-ului este polinomială, ordinul polinomului fiind dat de, aproximativ, numărul de biţi cu care se codifică starea semiautomatului echivalat. Rezultatul este încurajator. Ne putem aştepta ca transformarea reţelei de circuite de tip *formulă* într-una de tip *circuit* să aducă o simplificare notabilă, fără să presupună creşterea adîncimii peste $D(n) = O(\log n)$.



**Propoziţia 5.15** Complexitatea unei reţele combinaţionale de tip circuit asociată unui semiautomat finit cu $a$ stări este

$$S_C(n) = O(n)$$

dacă

$$D_C(n) = O(\log n). \square$$

*Demonstraţie* Pornind tot de la *Exemplul 5.36* se observă că

$$S^*_{MUX}(n) = S^*_{MUX}(\frac{n}{2})2 + (a(\frac{n}{2} - 1) + 1)S_{MUX}$$

cu

$$S^*_{MUX}(1) = 0,$$

iar

$$S^*_{CLC}(n) = S_{CLC}((n-1) \cdot a + 1).$$

Rezultă:

$$S_C(n) = S^*_{MUX}(n) + S^*_{CLC}(n) =$$
$$= (a(n-1) - (a-1)\log n)S_{MUX} + ((n-1)a + 1)S_{CLC}.$$

Deci:

$$S_C(n) = O(n). \square$$

Rezultatul este deosebit de favorabil deoarece am obţinut un polinom de ordin minim (unu) păstrînd o adîncime logaritmică. Circuitul este proporţional şi cu numărul de stări ale automatului, deci este mai complex decît cel din figura 5.110*a*, dar am scăzut timpul de execuţie de la $T(n) = O(n)$ la $T(n) = O(\log n)$.

Acest nivel de complexitate dă o speranţă celor ce gîndesc realizarea unor funcţii de prelucrare a şirurilor, de un anumit tip, cu ajutorul circuitelor combinaţionale. Căutarea unor soluţii ce presupun un timp de execuţie mai mic de $O(\log n)$ nu este justificată atît timp cît accesul celor $n$ simboluri la intrarea circuitului combinaţional nu poate fi realizat sub $T(n)$, datorită caracterului inerent secvenţial pe care îl au, deocamdată, tehnologiile de manipulare a şirurilor de simboluri. "Percepţia globală", nesecvenţială, nu caracterizează încă tehnologiile de transfer a informaţiei la, în şi de la sistemele digitale. Deci, nu are rost să ne grăbim prea tare cu prelucrarea şirurilor de $n$ simboluri dacă transferarea lor nu poate fi făcută decît tot într-un timp ce depinde proporţional de $n$.

### 5.9.3 Structură combinaţională generală asociată unui semiautomat finit

Reţeaua combinaţională asociată unui semiautomat finit este realizată cu două tipuri de structuri: multiplexoare şi *CLC*-uri. Primele au o configuraţie independentă de funcţionarea concretă a semiautomatului originar pentru $n$ şi $a$ date. Numai *CLC*-urile depind de semiautomatul concret de la care se porneşte. Deci, avem o substructură generală, trunchiul arborelui format din *MUX*-uri, şi una particulară, frunzele arborelui formate din *CLC*-uri. Care este ponderea cantitativă a celor două substructuri?

O primă observaţie pe care o putem face este aceea că un *CLC*, pe intrarea căruia se aplică o *variabilă* de intrare şi *valoarea* unei stări, devine o structură ce se poate simplifica semnificativ datorită valorii concrete ce se impune. Variabila $x_i$ are o valoare imprevizibilă la intrarea unui *CLC* (figura 5.109*a*), dar variabila $q_j$ este concretizată într-o reţea de tipul celor din figura 5.110.

**Exemplul 5.37** Semiautomatul din *Exemplul 5.18* într-o primă variantă are un *CLC* definit prin următoarele funcţii de tranziţie

$$Q_1^+ = Q_0,$$
$$Q_0^+ = Q'_1 + Q_0 A.$$

Cum arată *CLC*-ul pentru $Q_0 = 0$ iar $Q_1 = 1$? Rezultă că

$$Q_1^+ = "0",$$
$$Q_0^+ = "0",$$

deci *CLC*-ul se reduce la un circuit care generează pe ieşiri, independent de intrări, valoarea logică zero. Adică, intrările circuitului comandat de el sînt conectate la masă. Dar pentru $Q_0 = "1"$ şi $Q_1 = "1"$? Vom obţine:

$$Q_1^+ = "1",$$
$$Q_0^+ = A,$$

ş.a.m.d. □

**Exemplul 5.38** Dacă *CLC*-ul este descris prin

$$Q_1^+ = Q_1 A + Q'_0 B,$$
$$Q_0^+ = Q_1 Q_0 A,$$

atunci, pentru $Q_1 = "1"$ iar $Q_0 = "0"$ obţinem

$$Q_1^+ = A + B,$$
$$Q_0^+ = "0". \square$$

Remarcăm faptul că dacă ieşirea *CLC*-ului depinde de un singur bit aplicat pe intrarea $X$, atunci pentru un $q_j$ dat, *CLC*-ul nu presupune decît cel mult un circuit de inversare pentru bitul în cauză. Dacă *CLC*-ul are ieşiri care pot depinde, pentru



o anumiă stare, de doi sau mai mulţi biţi de intrare, atunci, pentru un $q_j$ dat, *CLC*-ul redus mai poate presupune circuite logice cu două sau mai multe intrări.

**Propoziţia 5.17** *CLC*-ul asociat unui semiautomat ce primeşte la intrare o secvenţă de $n$ simboluri poate fi realizat numai cu o reţea de *MUX*-uri, dacă din orice stare a semiautomatului tranziţia se realizează prin testarea a cel mult unui singur bit din configuraţia binară aplicată pe intrarea $X$. (Altfel spus: dacă semiautomatul are *reacţie negativă maximă* (conform *Definiţiei 5.14*) atunci reţeaua combinaţională asociată se reduce la reţeaua de multiplexoare.) Deci:

$$S_F(n) = S_{MUX}(n)$$

sau

$$S_C(n) = S^*_{MUX}(n). \square$$

*Demonstraţie* Structurile de tip *CLC* care formează frunzele reţelelor finale, de tipul celei din figura 5.110*e*, dispar complet dacă pentru orice $q_j$ ieşirile depind de un singur bit de intrare aplicat pe $X$. Acest lucru este evident datorită faptului că ieşirile de tip $Q_k^+$ devin *funcţii de o singură variabilă* deci presupun: conectarea directă a variabilei *(A)*, variabila negată *(A')*, conectarea la zero logic *("0")*, sau conectarea la unu logic *("1")*. Rezultă că este suficient ca fiecare bit ce formează configuraţiile binare de intrare $x_i$ să fie disponibil şi în varianta negată. □

Conform *Propoziţiei* ultime pentru un $n$ şi un $a$ date se poate concepe *un singur CLC*, sub forma unei reţele de *MUX*-uri, diferenţierea de la un caz particular la altul realizîndu-se numai prin modul de conectare al intrărilor *MUX*-urilor de pe ultimul nivel (cel al frunzelor) la variabilele binare ce codifică variabilele de intrare, la negatele acestora, la zero sau la unu logic. Care este preţul plătit prin creşterea complexităţii?

Preţul plătit este dat de creşterea numărului de stări ale unui automat atunci cînd acesta este transformat într-unul cu reacţie negativă maximă. Conform cu *Propoziţia 5.2* (paragraful 5.6.1) pentru a obtine un circuit combinaţional cu o structură identică, indiferent de natura semiautomatului cu $a$ stări ce recepţionează un şir de $n$ simboluri, va trebui să luăm în consideraţie un circuit combinaţional asociat unui semiautomat cu $a^* = a^2\text{-}a$ stări. Preţul plătit va fi o creştere a complexităţii de *(a-1)* ori. Într-adevăr:

$$S_C^*(n) = S_{MUX}((a^2\text{-}a)(n\text{-}1) + (a^2\text{-}a\text{-}1)\log_2 n)$$

pentru structura generală de circuit, iar

$$S_C^*(n)/S_C(n) = ((a^2\text{-}a)(n\text{-}1)+(a^2\text{-}a\text{-}1)\log_2 n) / (a(n\text{-}1)+(a\text{-}1)\log_2 n) \approx a\text{-}1$$

pentru un $n > n_0$, unde $n_0$ este un număr suficient de mare.

Ţinînd cont de faptul că structura rezultată este una generală, independentă de problemă, creşterea de *(a-1)* ori a complexităţii este absolut tolerabilă ţinînd cont de simplificările de proiectare şi tehnologice astfel dobîndite.

- O problemă interesantă pe care o punem cititorului este aceea de a studia cum creşte $S_C(n)$ dacă se impune ca $D(n) = O(1)$.
- Am văzut în paragraful 4.3.3 că o memorie *DRAM* se poate asimila, în ultimă instanţă, cu o *formulă* de *MUX*-uri avînd o complexitate $S(n) = O(n)$, unde

$n$ este numărul de biţi stocaţi. Remarcăm faptul că un semiautomat finit ce recepţionează un şir de $n$ simboluri se poate echivala printr-un *CLC* având complexitatea $S(n) = O(n)$ în varianta de tip *circuit*. Acest *CLC* este tot o reţea de multiplexoare. Funcţia de multiplexare se dovedeşte încă odată deosebit de semnificativă pentru domeniul sistemelor digitale.

• Care soluţie, cea cu semiautomat sau cea cu reţea de *MUX*-uri, se dovedeşte mai eficientă? Greu de răspuns. Am putea porni de la estimarea produsului $S(n) \cdot T(n)$, unde $T(n)$ este timpul necesar prelucrării, proporţional, evident, cu $D(n)$. Pentru soluţia secvenţială

$$S(n) \cdot T(n) = O(1) \cdot O(n) = O(n),$$

iar pentru cea combinaţională

$$S(n) \cdot T(n) = O(n) \cdot O(\log n) = O(n \log n).$$

Pentru ambiţia de a creşte viteza de calcul am fost puternic penalizaţi prin creşterea complexităţii într-o mai mare măsură.



# 6

# PROCESOARE
## Sisteme de ordinul trei

Conform principiului enunţat în capitolul *2*, trecerea la o clasă de sisteme cu o funcţionalitate mai complexă se realizează prin introducerea unei bucle de reacţie suplimentare. Vom conecta, deci, într-o buclă un automat finit *A*, cu un alt sistem logic ce are cel mult ordinul doi. Rezultă trei tipuri de sisteme de ordinul trei (*SO3*): cu bucla închisă printr-un *circuit combinaţional* (figura 6.1*a*), printr-o *memorie* (figura 6.1*b*) sau printr-un alt *automat* (figura 6.1*c*). Numai cea de a treia

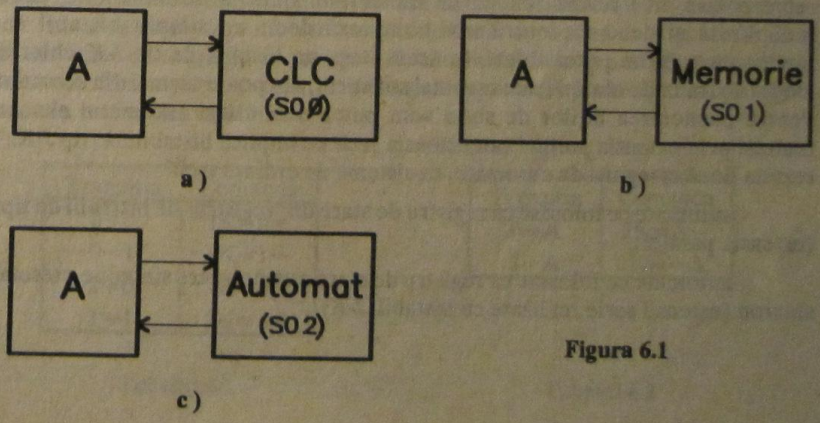



Figura 6.1

categorie se constituie în clasa *sistemelor de procesare* sau, altfel spus, în clasa *procesoarelor*. Ne permitem, printr-un uşor abuz de limbaj, să punem toate sistemele de ordinul trei sub incidenţa termenului de procesor, ţinînd cont de faptul că funcţia dominantă asociată acestor sisteme este cea de procesare.

Conectînd în bucla unui automat un *CLC* nu se poate depăşi nivelul funcţional al automatelor. Rezultă numai posibilităţi de optimizare a structurii circuitului combinaţional, cel ce intră sub incidenţa proiectării concrete.

Dacă în bucla unui automat apare un circuit de memorare se optimizează numai modul de prelucrare a stării interne. În esenţă, circuitul rămîne să efectueze tot o funcţie de automat finit. Deci, se obţine tot un efect de simplificare a structurii fizice şi de optimizare a procesului de proiectare.

Numai prin interconectarea a două automate vom obţine un salt funcţional notabil. În acest caz automatul *A* "colaborează" cu o structură ce are posibilităţi de acelaşi ordin pentru că este tot un automat. Din această interacţiune ce se produce de pe "poziţii echivalente" poate rezulta un comportament funcţional superior. Dacă *A+CLC* sau *A+Memorie* rămîn tot nişte automate, procesoarele (*A+Automat*) sînt mai mult decît nişte simple automate.

Vom aborda deci în primele două subcapitole, ce urmează, modalităţi de optimizare în proiectarea şi realizarea automatelor finite, urmînd ca numai apoi să trecem la tratarea unor structuri din categoria procesoarelor.

## 6.1 Automate, ca sisteme de ordinul trei

Pornim de la observaţia, elementară, că un automat finit presupune un circuit cu funcţia de memorare şi un circuit combinaţional ce calculează funcţiile de tranziţie (conform reprezentărilor din figura 5.31). Circuitului combinaţional din figura 6.1*a* îi vom asocia calculul funcţiilor de tranziţie, urmînd ca automatului, din aceeaşi reprezentare, să-i revină funcţia de memorare. Funcţia de memorare poate fi îndeplinită şi de o structură mai complexă decît un simplu bistabil *D* tip master-slave. Vom putea folosi, în acest scop, un bistabil de tip *J-K*, chiar dacă modul de tranziţie al stării sale este mai sofisticat, sau poate tocmai din acest motiv. Pentru memorarea biţilor de stare vom putea deci folosi automatul elementar realizat prin extensia *paralel* sau extensia *serie* ce implică bistabili de tip *J-K*. Vor rezulta două categorii de automate, ca sisteme de ordinul trei:

- automate ce folosesc ca registru de stare un "registru" de bistabili de tip *J-K* (extensia paralel);

- automate ce folosesc ca registru de stare numărătoare sincrone presetabile sincron (extensii serie realizate cu bistabili *J-K*).



### 6.1.1 Automate realizate cu "registre" de *J-K*-uri

La intrarea unui registru, format în mod normal din bistabili master-slave de tip *D*, se aduce în fiecare ciclu al ceasului configuraţia binară ce reprezintă starea următoare: $Q_n^+ Q_{n-1}^+ ... Q_0^+$. În fiecare ciclu de ceas, pe intrarea fiecărui bistabil de tip *D*, sîntem obligaţi să definim *valoarea bitului de stare*. Dacă vom substitui registrul cu un "registru" de *J-K*-uri va trebui să definim, pentru fiecare bistabil *J-K* de stare, comportamentul în fiecare stare *în funcţie de valoarea din starea curentă şi cea din starea următoare*. Fiecare bistabil *J-K* este un automat şi va fi tratat ca atare. Autonomia pe care o posedă ne va permite o comandă "mai puţin autoritară", în sensul că nu vom fi obligaţi să specificăm riguros, în fiecare moment (ciclu de ceas) modul în care variază cele două variabile de intrare, *J* şi *K*.

- **Regula de translatare a diagramelor de stare** pentru a permite implementarea cu "registre" de *J-K*-uri va ţine cont de faptul că pentru fiecare tranziţie se cunoaşte valoarea iniţială şi cea finală a tuturor biţilor de stare. Astfel, un bit de stare poate comuta din *0* în *0*, din *0* în *1*, din *1* în *0* sau din *1* în *1*. Cum va trebui să comandăm, în fiecare caz, bistabilul *J-K* pe cele două intrări?

Pentru comutarea *0→0* bistabilul poate fi comandat, la alegere, în două moduri. Poate primi comanda *J=K=0*, conform căreia nu comută, sau poate primi, în egală măsură, comanda *J=0*, *K=1*, conform căreia comută în zero. În ambele cazuri se va obţine acelaşi efect. Deci, comanda poate fi sintetizată prin *J=0*, *K=–*, în sensul că *J=0* iar *K* poate lua orice valoare, *0* sau *1*.

Pentru comutarea *0→1* bistabilul poate primi comanda *J=K=1*, prin care trece în starea complementară celei curente, sau comanda *J=1*, *K=0* prin care comută în *1*. Deci, sintetizînd, vom scrie *J=1*, *K=–*.

Pentru comutarea *1→0* bistabilul poate comuta în starea complementară (*J=K=1*) sau poate comuta în *0* (*J=0*, *K=1*). Deci, *J=–*, *K=1*.

Pentru tranziţia *1→1* se poate comanda *J=K=0* sau *J=1*, *K=0*, deci *J=–*, *K=0*.

Vom sintetiza cele anterioare în tabelul 6.1, din care vom putea deduce tabelul 6.2 pentru cazul în care tranziţia este definită printr-o *variabilă*, nu printr-o valoare. Ştim întotdeauna valoarea curentă a unei variabile binare de stare, dar uneori valoarea următoare nu este explicit definită, depinzînd de o variabilă. Este cazul tabelelor de tranziţie cu variabile de intrare incluse în ele. Pentru astfel de cazuri avem două tipuri de tranziţii: *0→A* şi *1→A*, unde *A* poate fi o funcţie logică de una sau mai multe variabile.

| Tranziţia | Comanda |
|---|---|
| 0→0 | JK=0– |
| 0→1 | JK=1– |
| 1→0 | JK=–1 |
| 1→1 | JK=–0 |

**Tabelul 6.1**

| Tranziţia | Comanda |
|---|---|
| 0→A | JK=A– |
| 1→A | JK=–A' |

**Tabelul 6.2**

• **Metoda de implementare** are ca etapă esenţial nouă translatarea diagramelor de tranziţie scrise pentru registre standard (realizate cu bistabile de tip $D$), în diagrame de tranziţie scrise pentru "registre" de $J$-$K$-uri. Unei diagrame pentru $D$-uri îi vor corespunde două diagrame, una pentru intrarea $J$ şi alta pentru intrarea $K$. Vom obţine dublarea numărului de funcţii logice pentru partea de semiautomat a automatului, fiecărui $Q_i^+$ îi va corespunde cîte un $J_i$ şi un $K_i$. În rest, algoritmul de implementare rămîne neschimbat.

**Exemplul 6.1** Reluăm semiautomatul ataşat automatului din *Exemplul 5.19* pentru a-l realiza ca sistem de ordinul trei. În figura 6.2*a* este dată funcţia $\lambda$, de tranziţie a semiautomatului. Aplicînd procedeul anterior expus, sprijiniţi pe tabelele 6.1 şi 6.2 se pot construi tabelele de tranziţie asociate implementării cu bistabili de tip $J$-$K$ (conform figurii 6.2*b*).

Pentru $J_1\,K_1$ vom completa diagramele după cum urmează:

- pentru tranziţia din starea *00*, $Q_1$ comută din *0* în *1*, deci vom completa în locaţiile *0*: $J_1=1$, $K_1=-$;

- pentru tranziţia din starea *01*, $Q_1$ comută din *0* în *0*, deci vom completa în locaţiile *1*: $J_1=0$, $K_1=-$;

- pentru tranziţia din starea *10*, $Q_1$ comută din *1* în *C*, deci vom completa în locaţiile *2*: $J_1=-$, $K_1=C'$ (conform tabelului 6.2);

- pentru tranziţia din starea *11*, $Q_1$ comută din *1* în *AB*, deci vom completa în locaţiile *3*: $J_1=-$, $K_1=(AB)'$.

Pentru $J_0\,K_0$ se adoptă aceleaşi procedee.

Urmează extragerea funcţiilor logice sub forma:

$$J_1 = Q'_0,$$
$$K_1 = Q_0\,(AB)' + Q'_0\,C' = Q_0A' + Q_0B' + Q'_0\,C',$$
$$J_0 = Q'_1,$$
$$K_0 = Q'_1 + (AB')' = Q'_1 + A' + B.$$

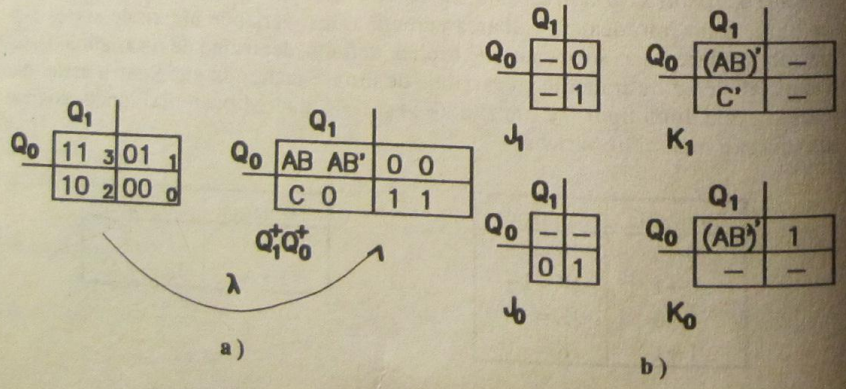



**Figura 6.2**



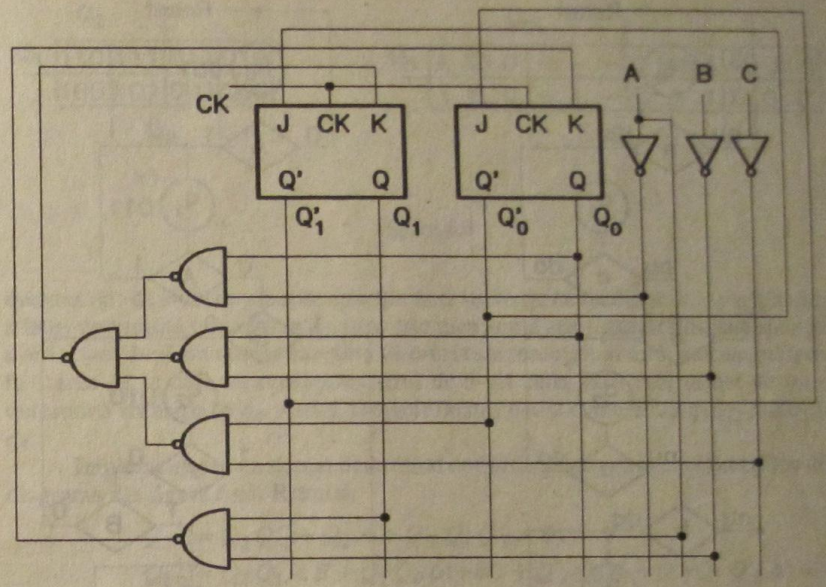



**Figura 6.3**

Structura semiautomatului astfel proiectat este reprezentată în figura 6.3. Dacă vom compara circuitul combinaţional rezultat cu $CLC_1$ din figura 5.53 vom vedea că, deşi am implementat patru funcţii logice în loc de două, am obţinut o structură mai simplă. □

**Exemplul 6.2** O aplicaţie tipică pentru semiautomate o constituie *recunoaşterea* apartenenţei unei secvenţe de simboluri la o mulţime de şiruri. Să considerăm mulţimea de şiruri

$$L = \{a^n b^m \mid n, m \geq 1\}.$$

Un element al şirului apare între simbolurile delimitatoare, *e*, sub forma

$$...eaa...abb...be...,$$

unde *a*-urile se repetă de *n* ori iar *b*-urile de *m* ori. Apariţia unui şir corect format poate fi semnalată prin starea finală a unui semiautomat iniţial, definit după cum urmează:

$$SA = (\{a,b,e\},\ \{q_0, q_1, ..., q_5\},\ \lambda\ ;\ q_0)$$

unde $\lambda$ este definită prin organigrama din figura 6.4. Codificarea elementelor mulţimii $X$, conform tabelului 6.3, ne permite redesenarea organigramei în varianta binară, în figura 6.5. Conform codificării propuse în această ultimă reprezentare se poate defini diagrama de tranziţie a semiautomatului (figura 6.6*b*).

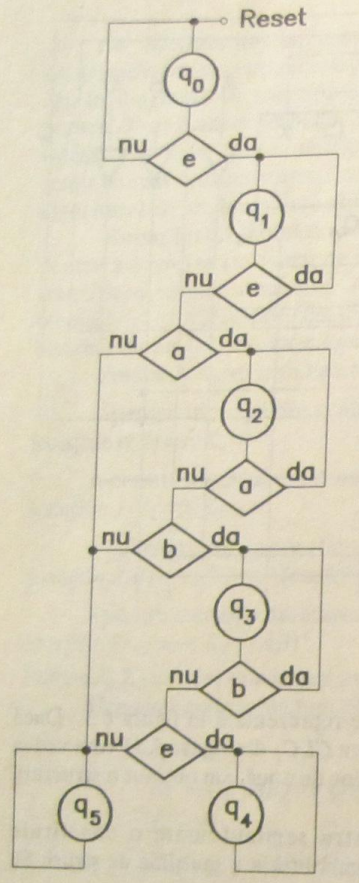



Figura 6.4

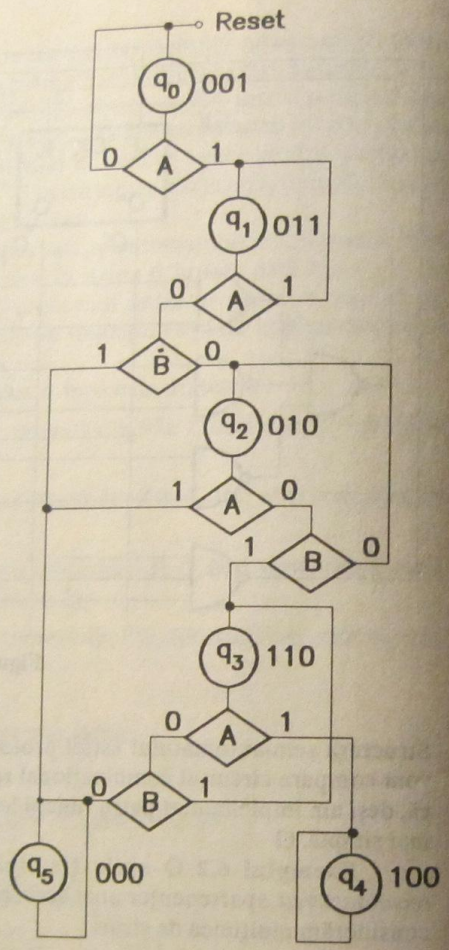



Figura 6.5

| A | B | |
|---|---|---|
| 0 | 0 | a |
| 0 | 1 | b |
| 1 | 0 | e |
| 1 | 1 | – |

Tabelul 6.3

Dacă automatul ajunge în starea $q_5$, atunci şirul ce urmează unui şir de *e*-uri nu aparţine mulţimii $L$. Dacă automatul se opreşte în $q_4$, şirul a fost format conform regulii ce defineşte elementele mulţimii $L$. În urma resetării, semiautomatul trece în starea $q_0$ în care aşteaptă apariţia unui *e*, care, odată recepţionat, determină tranziţia semiautomatului în $q_1$, stare în care este aşteptată terminarea unui



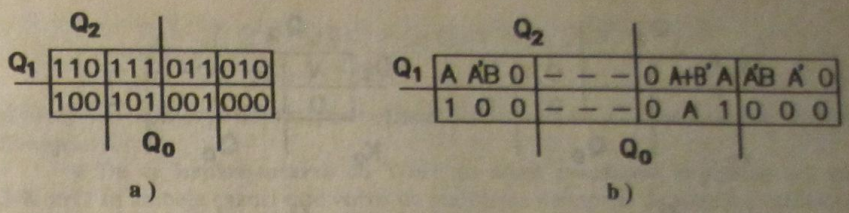



Figura 6.6

eventual şir de *e*-uri ce precede apariţia unei secvenţe de recunoscut. Apariţia unui *a* în $q_1$ determină tranziţia în $q_2$, care este menţinută atît timp cît sînt recepţionate *a*-uri. Dacă după un număr oarecare de *a*-uri este recepţionat un *b*, automatul trece în starea $q_3$ în care va recepţiona şirul de *b*-uri care, dacă este urmat de un *e*, determină tranziţia în $q_4$. Altfel, semiautomatul poate comuta din $q_1$, $q_2$ sau $q_3$ în $q_5$.

Implementarea ca sistem de ordinul doi presupune extragerea funcţiilor din diagrama din figura 6.6*b*. Rezultă:

$$Q_2^+ = Q_2 Q'_1 + Q_2 A + Q'_2 Q_1 Q'_0 A'B,$$
$$Q_1^+ = Q_2 Q_1 A'B + Q_1 Q_0 (A+B') + Q'_1 Q_0 A + Q'_2 Q_1 Q'_0 A' =$$
$$= Q_2 Q_1 A'B + Q_0 A + Q_1 Q_0 B' + Q'_2 Q_1 Q'_0 A',$$
$$Q_0^+ = Q'_1 Q_0 + Q_0 A.$$

(Funcţia $Q_1^+$ a presupus o minimizare suplimentară deoarece, aşa cum am arătat, reprezentările cu variabile sau funcţii în diagramă pot presupune, uneori, o astfel de operaţie.)

Merită să încercăm şi o implementare cu *J-K*-uri! Vom desena, în consecinţă, diagramele din figura 6.7, din care rezultă:

$$J_2 = Q_1 Q'_0 A'B,$$
$$K_2 = Q_1 A',$$
$$J_1 = Q_0 A,$$
$$K_1 = Q_2 (A'B)' + Q_0 (A+B')' + Q'_2 Q'_0 A =$$
$$= Q_2 A + Q_2 B' + Q_0 A'B + Q'_0 A,$$
$$J_0 = "0",$$
$$K_0 = Q_1 A'.$$

Complexitatea circuitului în primul caz este *35* iar în cel de al doilea, folosind bistabili *J-K* se reduce la *23*. □

• **De ce implementarea cu J-K-uri reduce dimensiunea CLC-ului?** Este o întrebare cu atît mai justificată cu cît numărul de funcţii ce trebuie implementate creşte. Faţă de implementarea cu bistabili de tip *D*, ele se dublează. Am putea încerca o justificare prin faptul că "registrul" posedă autonomie proprie în virtutea căreia poate evolua şi sub un control diminuat din partea buclei combinaţionale.

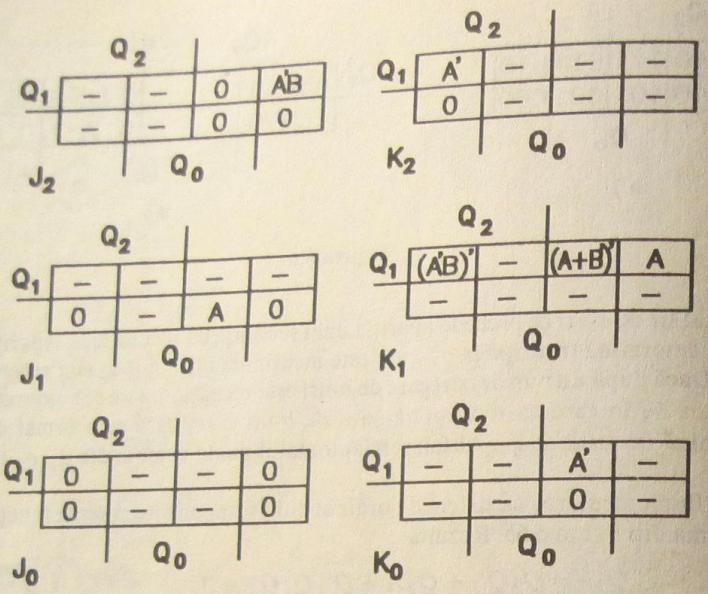



**Figura 6.7**

Răspunsul este oarecum justificat dar, prin exemplul ce urmează, vom încerca să arătăm că este necesară o argumentaţie suplimentară.

**Exemplul 6.3** Să încercăm implementarea semiautomatului din *Exemplul 6.1* cu ajutorul unui "registru" de *T*-uri. Şi bistabilii de tip *T* pot folosi la structurarea memoriei de stare a unui semiautomat. Tabelele după care poate fi determinată comanda unui bistabil de tip *T*, astfel încît să poată servi ca element de memorare pentru un bit de stare, sînt reprezentate în tabelul 6.4 şi în tabelul 6.5. Evident, dacă valoarea bitului de stare comută, atunci *T=1*, altfel, *T=0*. Din primul tabel se deduce imediat conţinutul celui de al doilea. Pornind de la diagramele de tranziţie din figura 6.2*a* se deduc cele din figura 6.8 conform regulilor date prin tabelele 6.4 şi 6.5. Din aceste diagrame extragem următoarele ecuaţii:

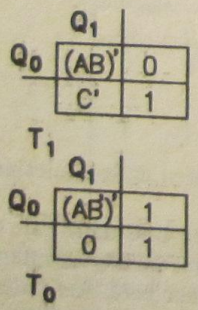



**Figura 6.8**

| Tranziţia | Comanda |
|---|---|
| 0→0 | T=0 |
| 0→1 | T=1 |
| 1→0 | T=1 |
| 1→1 | T=0 |

**Tabelul 6.4**

| Tranziţia | Comanda |
|---|---|
| 0→A | T= A |
| 1→A | T=A' |

**Tabelul 6.5**



$$T_1 = Q'_1 Q'_0 + Q'_0 C' + Q_1 Q_0 A' + Q_1 Q_0 B',$$
$$T_0 = Q'_1 + Q_0 A' + Q_0 B,$$

şi observăm că, nici pe departe, nu obţinem efectul remarcat cu ocazia rezolvării din *Exemplul 6.1*. □

● **De ce implementarea cu T-uri nu oferă avantajele implementării cu J-K-uri?** În ambele cazuri este vorba de realizarea memoriei de stare cu automate elementare. De ce autonomia *T*-ului nu este la fel de utilă ca cea a *J-K*-ului? Am putea răspunde acestei întrebări analizînd natura prelucrărilor logice realizate pe bucla celor două automate elementare. În bucla *J-K*-ului se află *AND*-uri iar în bucla *T*-ului se află un *XOR*. Poarta *AND* realizează o *funcţie pur logică*, pe cînd circuitul *XOR* are o puternică tentă *numerică*. Diversitatea comportamentului la ieşirea unui *AND* este mult atenuată faţă de dinamica intrărilor. Un *AND* cu opt intrări are *1* pe ieşire numai într-un caz din *256*, deci am putea spune că ieşirea lui este, cu o foarte mare probabilitate, egală cu *0*. Prelucrarea introdusă de un *AND* este foarte "puternică". Spre deosebire de poarta *AND*, circuitul *XOR* prezintă la ieşire o dinamică a semnalului comparabilă cu dinamica oricăreia dintre intrări.

Formal, am putea susţine cele anterioare utilizînd conceptul de *entropie* asociat unui semnal. Dacă la intrarea unui *CLC* semnalele logice variază nerestricţionat, atunci probabilitatea de apariţie a valorii *0*, $p_0$, este egală cu probabilitatea de apariţie a valorii *1*, $p_1$. În consecinţă fiecare intrare este caracterizată prin entropia

$$H_I = -\frac{1}{2}\log_2\frac{1}{2} - \frac{1}{2}\log_2\frac{1}{2} = 1.$$

Analizînd tabelul de adevăr pentru *XOR* observăm că la ieşire, dacă intrările sînt nerestricţionate, $p_0 = p_1$, atunci $H_{XOR} = 1$, entropia pe ieşire fiind aceeaşi cu cea a oricărei intrări.

Ce se întîmplă cu un *AND* cu *n* intrări? În acest caz în $2^n$-*1* cazuri ieşirea este *0* şi într-unul singur este *1*. Deci

$$H_{AND} = -\frac{2^n-1}{2^n}\log_2\frac{2^n-1}{2^n} - \frac{1}{2^n}\log_2\frac{1}{2^n}.$$

Pentru *n* suficient de mare, putem scrie

$$H_{AND} \approx \frac{n}{2^n},$$

entropia la ieşirea unui *AND* tinzînd, astfel, la valori mai mici, sau mult mai mici, decît cea de pe intrări.

Circuitul *AND* "face o ordine" mult mai mare decît circuitul *XOR*, capacitatea lui de prelucrare logică este mult mai mare decît cea a unui circuit aritmetic precum *XOR*-ul. Pe bucla de ordinul doi a *J-K*-ului se produce o prelucrare logică mult mai puternică decît pe bucla, de acelaşi ordin, a bistabilului de tip *T*. Diversitatea la intrarea în bucla *T*-ului (intrarea *XOR*-ului), este aceeaşi

cu cea de la ieşirea din buclă (ieşirea *XOR*-ului). Pe bucla bistabilului de tip *T* nu se produce nici un fel de prelucrare logică în sens informaţional; diversitatea nu este afectată. Acesta este, credem, motivul pentru care folosirea acestui tip de automat elementar, ca celulă de memorare pentru un bit de stare, nu-şi găseşte justificarea.

• **O explicaţie, mai pragmatică**, poate fi dată prin faptul că cel puţin jumătate din locaţiile diagramelor de tranziţie sînt completate cu o liniuţă în cazul implementării cu *J-K*-uri. Acest tip de bistabil nu trebuie comandat atît de "autoritar" ca celelalte, de tip *D* sau *T*; autonomia pe care o are, în virtutea faptului că este un automat finit, permite comenzi mai "vag" formulate, de tip *1–*, *0–*, sau altele asemenea.

• **Atenţie!** În cazul unor semiautomate foarte simple, reducerea complexităţii *CLC*-ului nu compensează creşterea în complexitate pe care o comportă realizarea unui bistabil de tip *J-K* faţă de unul de tip *D*. Vor fi, deci, cazuri, foarte simple, în care folosirea bistabililor de tip *D* să fie de preferat. Numai de la o anumită complexitate în sus realizarea automatelor cu *J-K*-uri este cu certitudine justificată.

Atunci cînd *CLC*-ul este realizat cu un *PLA* orientarea către soluţii de genul celor propuse în acest paragraf este obligatorie. În acest sens există şi circuite standard de tipul celui din figura 6.9, ce permit realizarea unor automate finite cu pînă la $2^n$ stări. Şansa ca, folosind un *PLA* cu un număr redus de produse elementare, să poată fi realizate un număr cît mai mare de automate distincte, este simţitor crescută prin folosirea *J-K*-urilor pe buclă.

• **Iarăşi, atenţie!** Dacă, *CLC*-ul pe bucla unui automat este realizat cu un circuit de tip *ROM*, atunci folosirea *J-K*-urilor devine o absurditate. Motivul este evident. Pentru cine nu este evident, se va gîndi cu atenţie.

• În capitolul anterior am studiat două tipuri de bistabili *J-K* (paragraful 5.22). Unul semitransparent pe palierul anterior frontului activ (figura 5.14) şi unul activat exclusiv de frontul activ al ceasului (figura 5.15). Calculul timpilor de

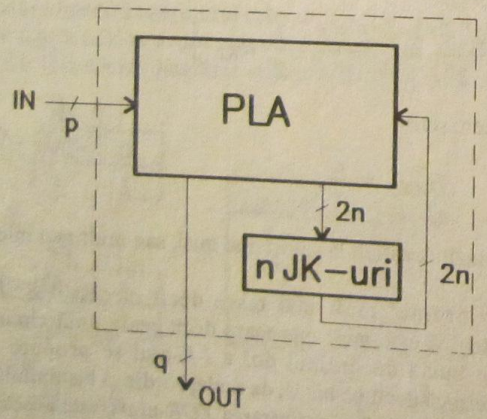



**Figura 6.9**



propagare pe buclă se va face diferit funcţie de bistabilul folosit. Perioada minimă a ceasului, *T*, va fi calculată în primul caz cu relaţia

$$T_{min} = t_{pJK} + t_{pCLC} + t_{SU} + T_1,$$

unde $T_1$ este durata palierului anterior frontului negativ, iar în cel de al doilea caz cu relaţia

$$T_{min} = t_{pJK} + t_{pCLC} + t_{SU}.$$

Este evident dezavantajul bistabilului semitransparent. El va putea fi folosit atunci cînd nu se impun restricţii de viteză, caz în care ne putem asigura că pe toată durata palierului ce precede frontul activ intrările *J* şi *K* au valori stabile.

## 6.1.2 Automate realizate cu numărătoare sincron presetabile

Dacă în cazul "registrului" din *J-K*-uri foloseam autonomia fiecărui bistabil în parte, în cazul numărătorului vom putea folosi autonomia ansamblului ce se manifestă prin capacitatea de a evolua specific, prin numărare, într-un spaţiu al stărilor. Printr-o *abilă* codificare a stărilor, *uneori*, vom putea face astfel încît stările succesive să poată fi "calculate" prin numărare. Astfel, registrul de stare va putea fi înlocuit printr-un *numărător sincron presetabil* care va îndeplini funcţia de registru (fiind sincron presetabil) şi va putea prelua, parţial, efortul buclei combinaţionale în determinarea stării următoare. Vom numi acest numărător *numărător de stare*.

Schema de principiu pe care o propunem este reprezentată în figura 6.10, unde $CLC_1$ determină pentru fiecare stare, în funcţie şi de intrare, cum are loc tranziţia semiautomatului, prin numărare *(UP)* sau prin încărcare *(LOAD)*, iar $CLC_2$ calculează valoarea stării ce se încarcă în numărător atunci cînd tranziţia se produce prin încărcare. Ieşirea circuitului $CLC_2$ este nesemnificativă atunci cînd tranziţia are loc prin numărare. Acest fapt va determina, uneori, o *simplificare drastică* a acestui circuit. În locul circuitului $CLC_1$ vom putea conecta, de multe ori, un *circuit standard* simplu. Deci, *sperăm ca* $CLC_1$ *să fie standardizat iar* $CLC_2$ *simplificat*. Se impune, în acest moment, un exemplu.

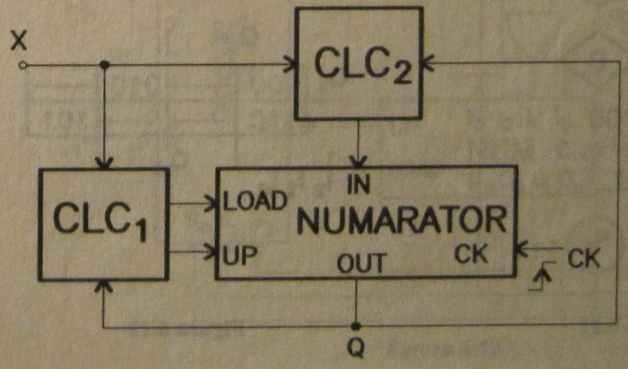



**Figura 6.10**

**Exemplu 6.4** Organigrama din figura 6.11 reprezintă un semiautomat cu trei intrări, *A*, *B* şi *C*, sincrone. Pentru a-l implementa conform procedeului anterior expus vom propune o astfel de codificare a stărilor încît un număr cît mai mare de tranziţii să se producă prin numărare binară. O variantă posibilă este marcată chiar pe figură. Din starea $q_0$ tranziţia se face prin numărare dacă *A=1*, altfel, numărătorul va fi presetat la valoarea *101*, corespunzătoare codificării stării $q_1$. Din $q_1$, $q_2$, $q_3$ şi $q_6$ tranziţia se face prin numărare. Din $q_4$ se comută prin numărare pentru *B=0*, iar din $q_5$ şi $q_7$ se comută numai prin presetarea numărătorului de stare. Vom putea determina, în consecinţă, valorile variabilelor de ieşire ale $CLC_1$ (figura 6.10) cu ajutorul unui multiplexor şi a unui inversor conectate conform reprezentării din figura 6.13. În fiecare stare, determinată de ieşirile $Q_2$, $Q_1$ şi $Q_0$ ale numărătorului, este selectată prin *MUX* cîte una din intrări pentru a determina comanda *UP* şi *LOAD = (UP)'*. Astfel în starea $q_0$, cînd $Q_2Q_1Q_0 = 000$, *W = UP = (LOAD)' = A*, determinînd numărarea dacă *A=1*, sau încărcarea numărătorului cu valoarea de pe intrările $I_2$, $I_1$ şi $I_0$. Intrările *1* şi *2* ale *MUX* sînt conectate la unu logic (*"1"*) astfel încît în stările $q_2$ şi $q_3$, codificate prin *001* şi *010*, numărătorul se va incrementa, tranziţia semiautomatului realizîndu-se, conform codificării, prin numărare. Mai observăm că în stările codificate prin *100* şi *110*

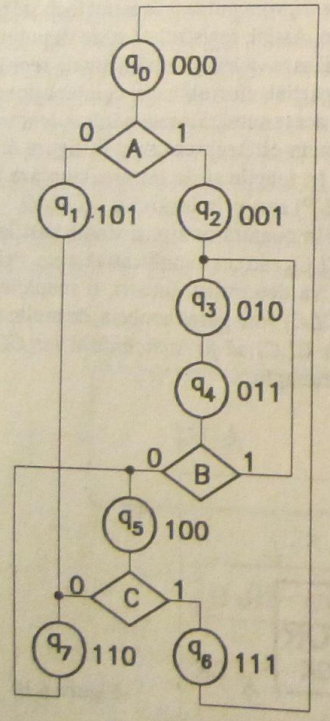



**Figura 6.11**

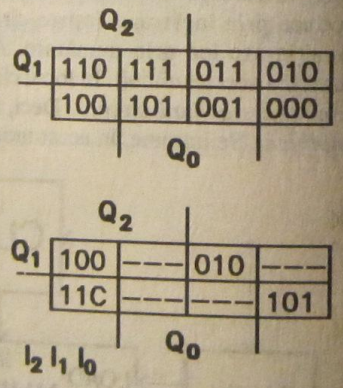



**Figura 6.12**



numărătorul se va preseta *(LOAD=1)*, sub comanda frontului activ al ceasului, la valorile găsite pe intrările $I_2$, $I_1$, $I_0$.

Pentru proiectarea circuitului $CLC_2$ (figura 6.10) vom defini diagrama de tranziţie din figura 6.12. Vom completa numai locaţiile corespunzătoare tranziţiilor care se realizează condiţionat sau necondiţionat prin presetarea (încărcarea) numărătorului. Deci, stărilor codificate cu *001*, *010*, *101* şi *111* nu trebuie să le definim valori precise pe intrările $I_2$, $I_1$, $I_0$. În aceste stări pe intrările de încărcare ale numărătorului se poate afla orice configuraţie binară deoarece cu certitudine: *UP=1* şi *LOAD=0*, tranziţia realizîndu-se prin incrementare. Patru locaţii din diagramă vor fi completate cu liniuţe, corespunzător unei valori indiferente. Acest fapt va contribui la simplificarea circuitelor logice asociate. Din diagrame rezultă:

$$I_2 = Q'_0,$$
$$I_1 = Q_0 + Q_2 Q'_1,$$
$$I_0 = Q'_2 Q'_1 + Q'_1 C.$$

Transformînd ecuaţiile cu teorema lui De Morgan rezultă circuitele reprezentate în figura 6.13.

Menţionăm, încă o dată, că *numărătorul trebuie să fie presetabil sincron*, încărcarea fiind comandată de frontul şi nu de palierul impulsului de ceas; numai astfel bucla prin $CLC_2$ (figura 6.10) se poate închide stabil. (Spre exemplu, numărătorul poate fi 74169). □

Observăm că $CLC_1$ a putut fi realizat *cu un circuit standard simplu*: multiplexorul. De asemenea $CLC_2$ a fost drastic simplificat deoarece în unele stări tranziţia semiautomatului a fost realizată prin numărare, această buclă nefiind

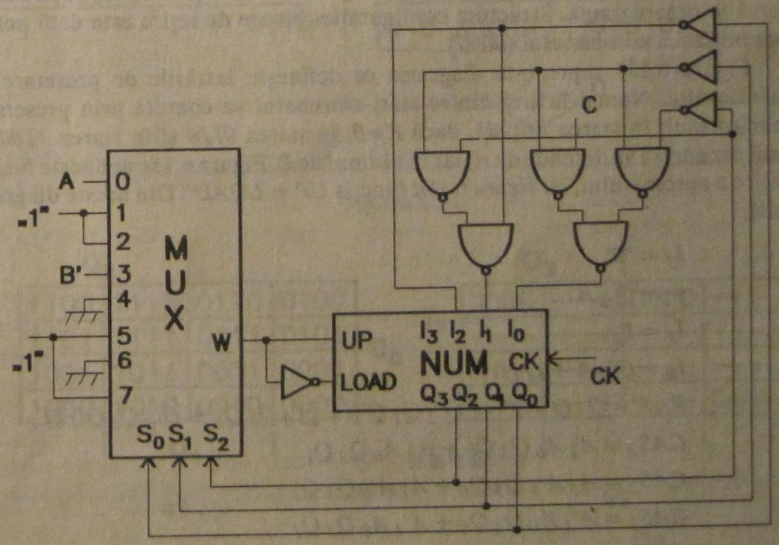



**Figura 6.13**

solicitată. Autonomia evoluției numărătorului degrevează bucla prin $CLC_2$ de o parte din efortul de control al evoluției semiautomatului. Bucla ce asigură numărarea permite simplificarea buclei prin $CLC_2$. Iarăşi, *o structură standard, simplu de proiectat şi de extins, se substituie parţial unei structuri ce depinde de problema concretă de proiectare rezolvată.*

• **Efectul variabilelor asincrone pe intrările UP şi LOAD** este disfuncţional pentru automatul sau semiautomatul proiectat prin acest procedeu. Acest fapt se justifică prin necesitatea de a asigura un timp de *Set-up* şi unul de *Hold*, al semnalului pe *UP* şi *LOAD*, faţă de frontul activ al ceasului *CK*. Deci există un interval de timp, în jurul tranziţiei active a *CK*, în care aceste două intrări trebuie să fie stabile pentru ca numărarea sau încărcarea să se efectueze corect.

Concluzionăm, că acest procedeu de implementare al semiautomatelor este restrictiv în sensul că *variabilele ce afectează intrările UP şi LOAD nu pot fi asincrone*. Printr-o codificare convenabilă (cu dependenţă redusă) variabilele ce afectează intrările de presetare ($I_i$) pot fi şi asincrone.

**Exemplul 6.5** În paragraful 4.3.3 am definit funcţional un *generator de ciclu* (*GC*) pentru o memorie *RAM* (figurile 4.30 şi 4.31). Proiectarea acestui circuit ar putea constitui un bun exemplu pentru metoda ce o propunem. Vom nota, simplificat, cu *A* semnalul *ACC* care semnifică faptul că se face *acces* la memorie. De asemenea, cu $A_1$ şi $A_0$ vom nota cei doi biţi de adresă ce sînt folosiţi pentru selectarea, prin $CAS_i$, a unui modul de memorie. Cu aceste notaţii şi pornind de la formele de undă din figura 4.31, se poate defini organigrama asociată automatului ce funcţionează ca generator de ciclu (*GC*). Această organigramă este reprezentată în figura 6.14. În starea codificată cu *0000* automatul (generatorul de ciclu) aşteaptă o comandă pentru declanşarea unui ciclu de regenerare (*R*) sau a unui ciclu de acces la memorie (*A*), pentru scriere sau citire. Regenerarea are prioritate, după cum se observă în organigramă. Structura configuraţiei binare de ieşire este dată pentru starea iniţială a automatului (*0000*).

Figura 6.15*b* reprezintă diagrama ce defineşte intrările de presetare ale numărătorului. Numai în trei dintre stări automatul va comuta prin presetarea numărătorului: în starea iniţială, dacă $R=0$, în starea *0110* şi în starea *1100*. În primul caz adresa va depinde de *A*, iar în ultimul de *R*. Figura 6.15*c* defineşte funcţia de ieşire a automatului, iar figura 6.15*d* funcţia $UP = LOAD'$. Din aceste diagrame rezultă:

$$I_3 = "0"$$
$$I_2 = Q'_2 A$$
$$I_1 = I_2$$
$$I_0 = Q'_2 A + Q_3 R$$
$$RAS = Q_3 Q'_2 Q'_1 + Q'_2 Q_1 Q'_0 + Q'_3 Q_1 Q_0 + Q'_3 Q_2 Q'_1 Q'_0$$
$$CAS_3 = A_1 A_0 Q_3 Q_0 + A_1 A_0 Q_3 Q_1$$
$$CAS_2 = A_1 A'_0 Q_3 Q_0 + A_1 A'_0 Q_3 Q_1$$
$$CAS_1 = A'_1 A_0 Q_3 Q_0 + A'_1 A_0 Q_3 Q_1$$
$$CAS_0 = A'_1 A'_0 Q_3 Q_0 + A'_1 A'_0 Q_3 Q_1$$



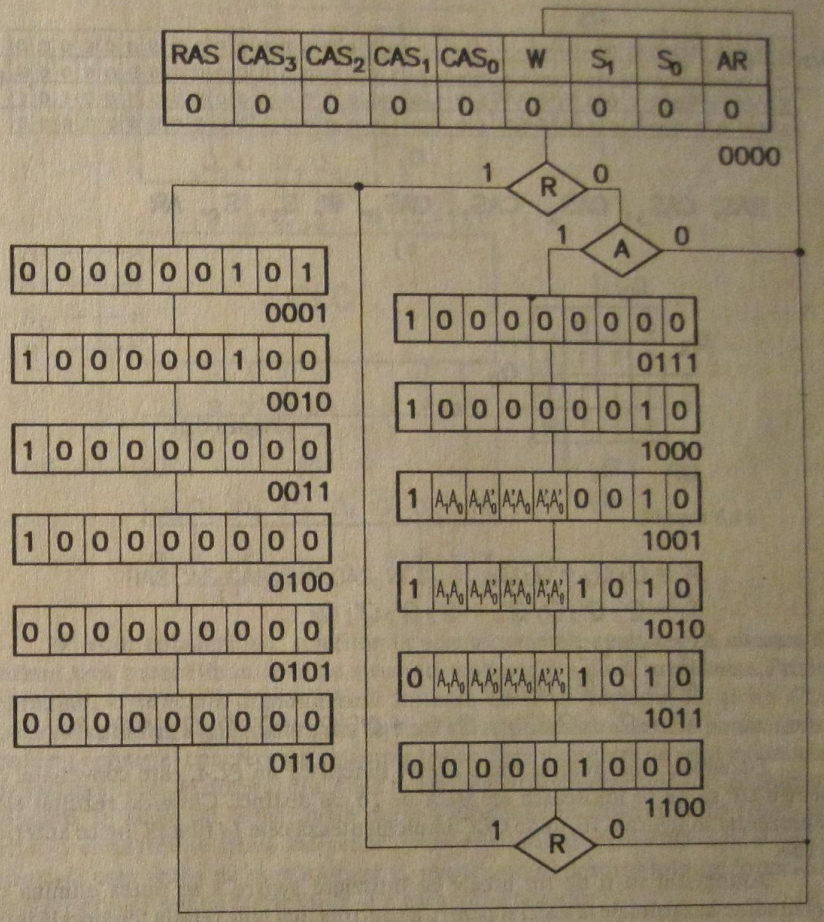



Figura 6.14

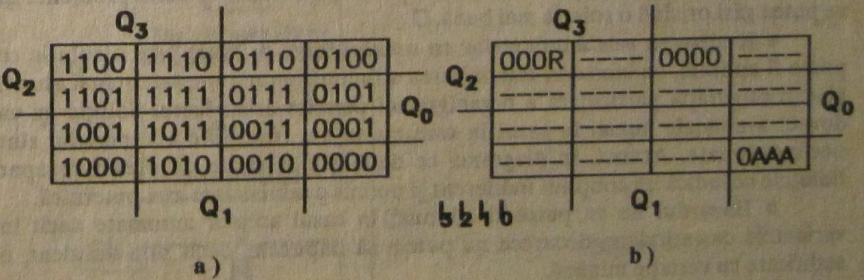
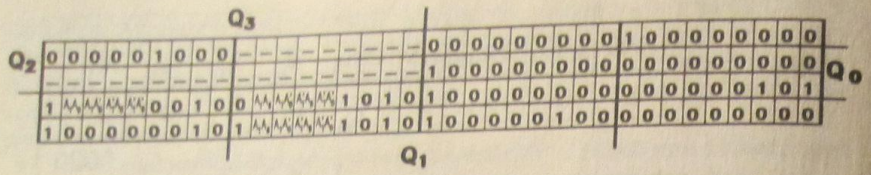
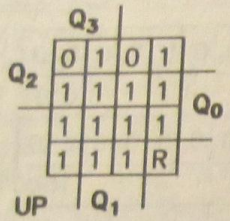



Figura 6.15




RAS, $CAS_3$, $CAS_2$, $CAS_1$, $CAS_0$, W, $S_1$, $S_0$, AR

c)




d)

**Figura 6.15**

$$W = Q_3 Q_2 + Q_3 Q_1$$
$$S_1 = Q'_3 Q'_2 Q_1 Q'_0 + Q'_3 Q'_2 Q'_1 Q_0$$
$$S_0 = Q_3 Q'_2$$
$$AR = Q'_3 Q'_2 Q'_1 Q_0$$
$$UP = Q_0 + Q_3 Q'_2 + Q_3 Q_1 + Q'_2 Q_1 + Q'_3 Q_2 Q'_1 + Q'_2 R.$$

Deoarece vom realiza toate aceste funcţii cu un *PLA*, este convenabil ca pentru *UP* să nu se folosească un *MUX* de *16* căi distinct. Circuitul rezultat este reprezentat în figura 6.16, unde *CLC* implementează cele *14* funcţii logice anterior scrise.

Automatul va fi de tip Mealy cu întîrziere pentru a se putea elimina cu certitudine hazardul de la ieşiri şi pentru a controla mai bine relaţia temporală între cele *9* semnale de ieşire.

Soluţia propusă este numai un exemplu de proiectare. Această problemă îşi va putea găsi oricînd o soluţie mai bună. □

• **Realizarea semiautomatelor cu numărătoare de stare** este o soluţie ce poate fi aplicată, cu efecte în minimizarea structurii unui automat, numai atunci cînd configuraţia particulară a organigramei permite acest lucru. Soluţia îşi va dovedi avantajele numai în cazul în care multe din tranziţiile automatului sînt necondiţionate. Atunci, în diagrama ce defineşte intrările de presetare apar liniuţele ce indică un conţinut indiferent şi permit o minimizare mai puternică.

• **Hazardul** nu va putea fi eliminat în cazul acestor automate decît în variantele cu întîrziere, deoarece nu putem să impunem, decît prin accident, o codificare cu variaţie minimă.



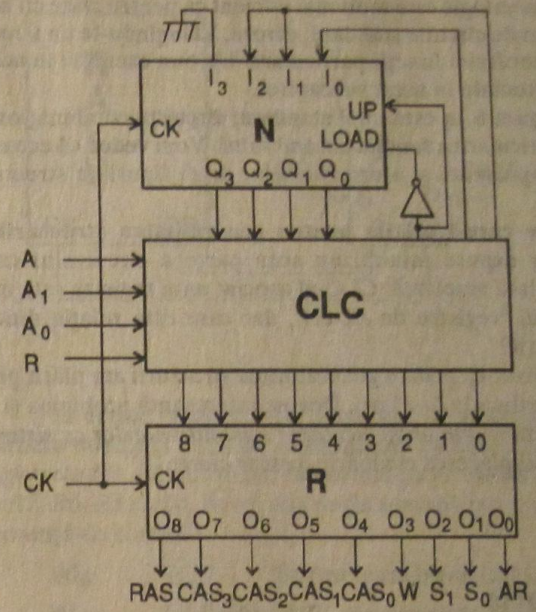



**Figura 6.16**

• Vom concluziona, referitor la **semiautomatele realizate ca sisteme de ordinul trei**, prezentîndu-le ca pe structuri **realizate din două subsisteme**. Primul reprezintă o *structură standardizată* formată dintr-un registru, *R*, şi un *CLC standard*, conform reprezentării din figura 6.17. Al doilea este un *CLC particularizat* conform definiţiei concrete a semiautomatului. Spre deosebire de un semiautomat realizat ca sistem de ordinul doi, unde numai registrul se constituie într-o structură standard, în acest caz o mai mare parte a efortului de structurare se realizează printr-o configuraţie fizică *independentă de problemă* şi *extensibilă după o regulă recursivă*; este vorba de numărătorul şi, uneori, multiplexorul folosite în metoda anterior expusă sau de bistabilii *JK* din paragraful anterior.

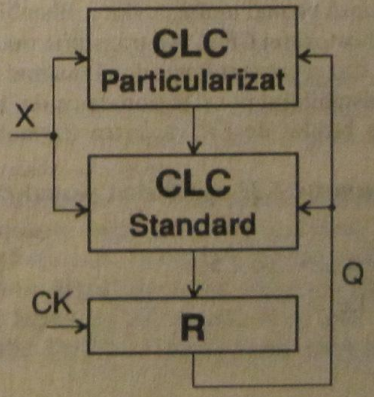



**Figura 6.17**

Din punct de vedere tehnologic este mult mai eficient ca pentru clase cît mai mari de probleme să dispunem de circuite standard, cărora, adăugîndu-le un număr cît mai mic de circuite, să le conferim funcţii particulare. Un bun exemplu în acest sens îl constituie structurile studiate în acest subcapitol.

Deci, automatul din figura 6.1*a* este unul standard, circuitul combinaţional din buclă avînd rolul de a particulariza funcţia ansamblului. Vom vedea că aceasta este caracteristica cea mai importantă şi a următoarelor două tipuri de structuri reprezentate în aceeaşi figură.

• **Care este preţul pe care-l plătim pentru generalitatea structurilor folosite?** Procedeele anterior expuse minimizau acea parte a circuitului care depindea de problema particulară rezolvată. *CLC*-ul asociat unui registru este mai complex decît cel asociat unui "registru de *JK*-uri", dar care este relaţia dintre complexităţile celor două soluţii?

În paragraful 5.9.3 am văzut că pentru generalitatea structurii am plătit prin creşterea complexităţii de aproximativ *(a-1)* ori. Devine interesantă problema şi în cazul *creşterii generalităţii structurii prin implementarea semiautomatelor ca sisteme de ordinul trei*. Lăsăm cititorului plăcerea evaluării acestor efecte.

## 6.2 Automate cu memorie

A doua categorie de sisteme de ordinul trei o constituie automatele ce funcţionează cuplate în buclă cu o memorie (figura 6.1*b*), mai precis spus, automatele ce au în buclă un sistem de ordinul unu, deoarece o funcţie de memorie poate fi realizată şi cu un sistem de ordin superior lui unu.

Vom începe prezentînd un exemplu pentru a ne putea forma o imagine mai concretă asupra acestei categorii de maşini.

**Exemplul 6.6** Dacă între ieşirea *CR* dintr-un *RALU* şi intrarea *CRIN* a aceluiaşi sistem se conectează un bistabil de tip *D* se obţine un sistem de ordinul trei cu facilităţi aritmetice îmbunătăţite (figura 6.18), în sensul că se pot realiza operaţii cu numere într-o reprezentare extinsă pe mai multe cuvinte. Bistabilul de tip *D* va reţine, în fiecare ciclu, valoarea indicatorului *CR* şi-l va transmite operaţiei din următorul ciclu, dacă numerele sumate sînt reprezentate pe două sau mai multe cuvinte. Intrările de *S*(set) şi *R*(reset) ale bistabilului pot fi acţionate cu doi biţi de comandă *SCR* şi *RCR* prin care valoarea bitului de *CR* va putea fi setată sau resetată.

Vom completa definiţia dată în *Exemplul 5.33* specificînd parţial cîmpul *<Alte Com>* după cum urmează:

*<Alte Com>* ::= *<Com.CR>*< ... > ... < ... >,
*<Com.CR>* ::= *SCR* | *RCR* | –.



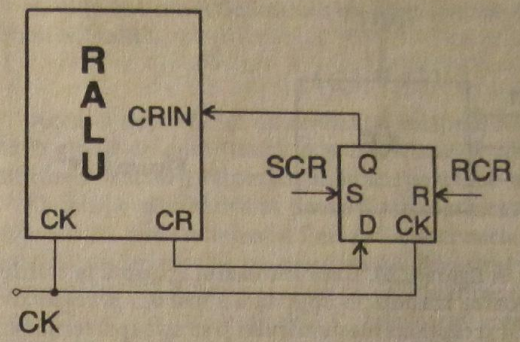



**Figura 6.18**

În aceste condiţii vom putea scrie o secvenţă de comenzi ce sumează două numere întregi şi pozitive reprezentate pe *64* de biţi. Primul număr este stocat în registrele *R3*, *R2*, *R1* şi *R0*, cu cel mai puţin semnificativ cuvînt în *R0*, iar al doilea în *R7*, *R6*, *R5* şi *R4*, cu cel mai puţin semnificativ cuvînt în *R4*. Rezultă următoarea secvenţă de comenzi:

$00_H$ *RCR* /Se resetează bistabilul de tip *D*/
$01_H$ *L0 R4 PLUS W* /Se sumează cuvintele ce reprezintă partea cea mai puţin semnificativă/
$02_H$ *L1 R5 PLUS W*
$03_H$ *L2 R6 PLUS W*
$04_H$ *L3 R7 PLUS W.*

În final în *R7*, ..., *R4* şi bistabilul *D* se află rezultatul operaţiei. Evident, cel mai semnificativ bit al rezultatului se află în bistabilul de tip *D*. □

• **Autonomia sistemului anterior structurat a fost crescută** faţă de cea a unui simplu *RALU*. Operaţia de sumare exemplificată nu ar fi putut fi realizată cu maşina din *Exemplul 5.33* printr-o secvenţă de *comenzi imperative*, în sensul că evoluţia secvenţei de comenzi este independentă de rezultatul operaţiilor comandate. Ar fi trebuit ca în urma fiecărei sume parţiale să fie testat indicatorul *CR* şi, în funcţie de rezultatul testării, să fie dată o comandă de sumare cu *CR=0* sau *CR=1*. Bucla fizică suplimentar introdusă ne scuteşte de această opţiune oferind o structură ce poate fi imperativ comandată.

Bucla suplimentară, ca şi în cazurile anterioare, a permis creşterea autonomiei sistemului.

**Exemplul 6.7** Vom considera automatul finit iniţial, *AF*, care prin starea în care ajunge, după ce a recepţionat un şir de numere pe calea *N*, indică valoarea maximă sesizată. În figura 6.19 este reprezentat acest automat care primeşte, de la un sistem digital oarecare, elemente din $N = \{ 0, 1, 2 \}$, semnalul *R* prin care se indică faptul că pe intrarea *N* se află o valoare generată de sistemul exterior şi semnalul *INIT* care iniţializează automatul. Organigrama ce descrie funcţionarea

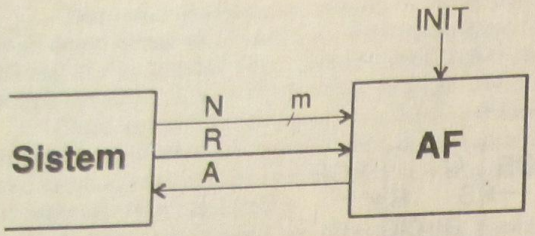



Figura 6.19

automatului este reprezentată în figura 6.20. Dacă automatul aşteaptă în stările *0*, *1* sau *2* atunci înseamnă că valoarea maximă recepţionată a fost *0*, *1* şi respectiv *2*. În stările *0'*, *1'* şi *2'* anunţă prin *A* recepţionarea numărului şi se aşteaptă terminarea semnalului *R* (care nu poate dispare decît ca o consecinţă a semnalului de achitare *A*).

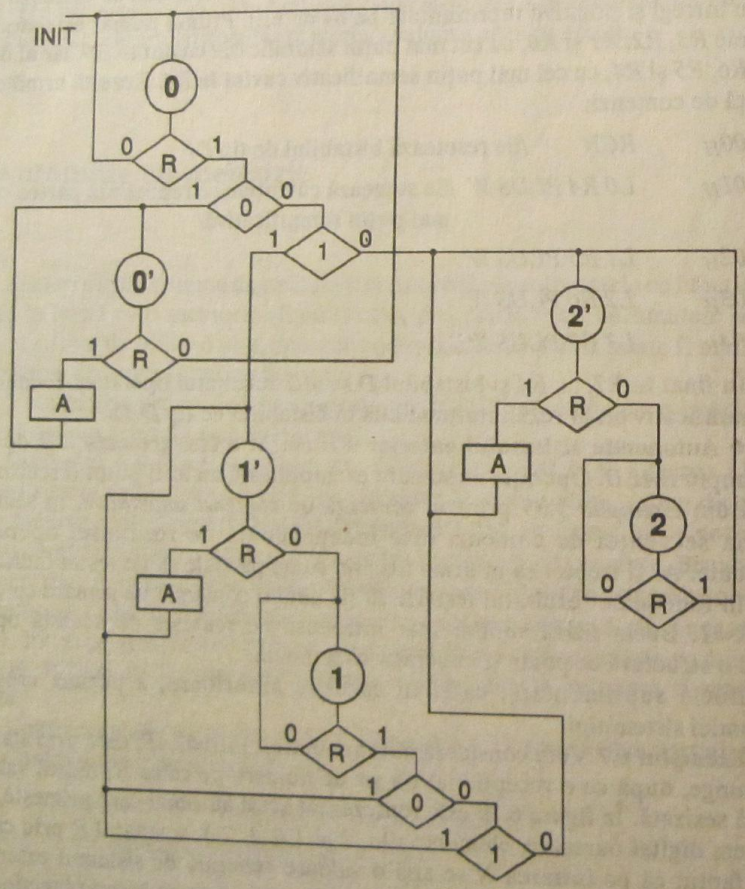



Figura 6.20



Automatul astfel definit are un spaţiu al stărilor ce depinde de cardinalul mulţimii *N*. Astfel,

$$|Q| = 2|N|.$$

Această dependenţă de cardinalul mulţimii *N* face ca automatul să nu mai poată fi gîndit ca unul "finit" în sensul clasic al termenului. Dacă $|N|$ creşte modificarea structurii automatului nu se poate face după o "regulă simplă şi clară."

O soluţie alternativă ar putea fi cea reprezentată prin schema de principiu din figura 6.21, unde automatul finit *AF'* este conectat cu un sistem de ordinul unu realizat cu un registru şi un comparator. Registrul primeşte comenzi de *LOAD* (încărcare) sau *CLR* (ştergere) de la *AF'*. Mărimea *N* de intrare este tot timpul comparată cu cea din registrul *R*, rezultatul comparării fiind testat de *AF'*. Organigrama ce descrie funcţionarea automatului *AF'* este dată în figura 6.22. În fiecare moment în *R* se găseşte valoarea maximă recepţionată.

Cardinalul mulţimii *N* determină capacitatea registrului şi comparatorului iar numărul de stări ale automatului finit este independent de această mărime. *AF'* este un "veritabil" automat finit. Această a doua soluţie ne permite să dimensionăm funcţie de $|N|$ numai acele resurse care cresc după o regulă recursivă simplă. □

• **Procesul de segregare în spaţiul stărilor** ce s-a produs (în exemplul anterior) prin definirea automatului *AF'* şi a registrului, pornind de la *AF*, este tipic

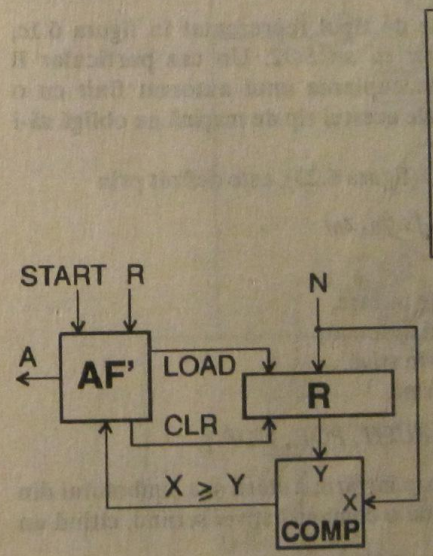



Figura 6.21

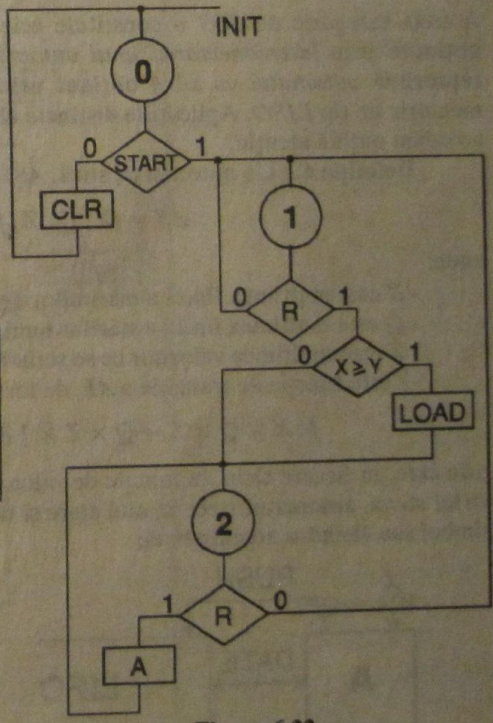



Figura 6.22

pentru degenerarea sistemelor de ordinul doi în sisteme de ordinul trei ca automate cu sisteme de memorie adiţională. Starea ansamblului este dată de starea automatului *AF'* şi de conţinutul registrului *R*, spre deosebire de prima soluţie în care starea sistemului era dată exclusiv de starea *AF*.

Spaţiul stărilor s-a structurat iar numărul de biţi a crescut. Într-adevăr: $|Q'|=3$ iar numărul de biţi ce descriu starea globală este $2+\lceil log_2 |N| \rceil$, faţă de $1+\lceil log_2 |N| \rceil$ pentru *AF*. Este preţul care, de regulă, se plăteşte pentru obţinerea unei structuri ce depinde de dimensiunea mulţimii de intrare printr-o regulă recursivă simplă.

• **Concluzionăm** spunînd că *SO3* realizate ca automate conectate în buclă cu un *SO1* permit, în principal:

- creşterea autonomiei sistemului, prin bucla suplimentară;
- o mai simplă dependenţă a complexităţii sistemului de cardinalul mulţimii de intrare.

## 6.3 Automatul cu stivă

A treia categorie de *SO3* o constituie cele de tipul reprezentat în figura 6.1c, obţinute prin *interconectarea unui automat cu un SO2*. Un caz particular îl reprezintă *automatul cu stivă* obţinut prin cuplarea unui automat finit cu o memorie de tip *LIFO*. Aplicaţiile distincte ale acestui tip de maşină ne obligă să-i acordăm puţină atenţie.

**Definiţia 6.1** Un automat cu stivă, *AS*, (figura 6.23), este definit prin

$$AS = (X, Q, Z, f; q_0, z_0)$$

unde:

- $X$ este mulţimea finită a mărimilor de intrare,
- $Q$ este mulţimea finită a stărilor unui automat;
- $Z$ este mulţimea valorilor ce se scriu în stivă,
- $f$ este funcţia de tranziţie a *AF*, de forma

$$f: X \times Q \times Z \rightarrow Q \times Z \times \{ PUSH, POP, NOP \},$$

prin care, în fiecare ciclu, în funcţie de valoare intrării, a stării şi a simbolului din vîrful stivei, automatul trece în altă stare şi dă o comandă stivei scriind, citind un simbol sau lăsînd-o neschimbată;

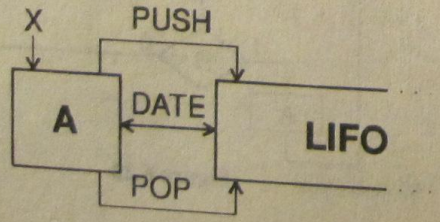



**Figura 6.23**



- $q_0 \in Q$ este starea iniţială a automatului,
- $z_0 \in Z$ este simbolul iniţial din vîrful stivei. □

Am arătat în paragraful 5.8.2 că o stivă este un automat finit. Deci un *AS* este un sistem de ordinul trei. Prin ce se dovedeşte superior unui automat finit elementar? Prin faptul că posedă o resursă dimensionabilă independent de complexitatea problemei rezolvate, automatul finit, şi o altă resursă ce poate fi oricît de mare deoarece creşte conform unei reguli simple, recursive. Un exemplu va face afirmaţia anterioară mai limpede.

**Exemplul 6.8** Să ne reamintim problema rezolvată în *Exemplul 6.2*, în care recunoşteam şiruri de tipul $a^n b^m$ cu $n, m \geq 1$. Ce probleme suplimentare ne-ar pune cazul, aparent particular, în care s-ar impune suplimentar ca $n=m$? Organigrama

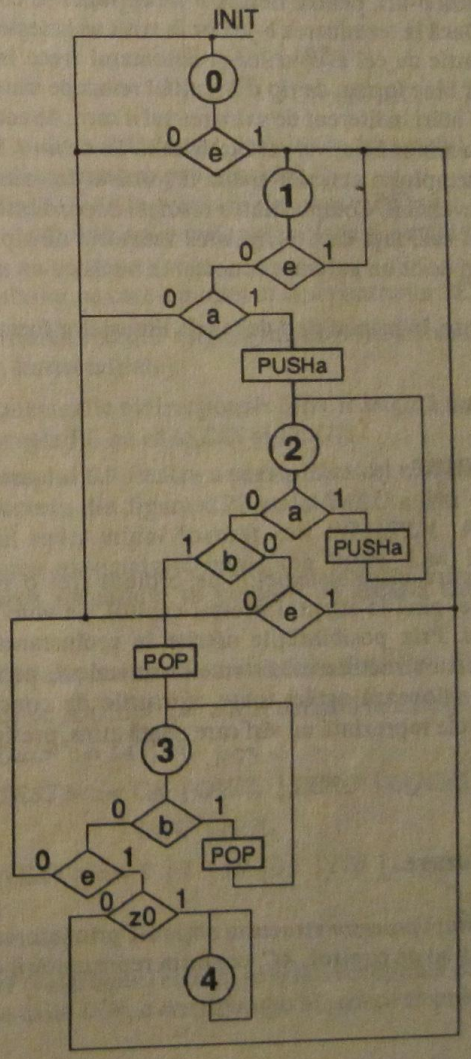



**Figura 6.24**

ce ar trebui concepută pentru un automat finit ce ar recunoaşte şiruri pentru care $n \leq n_0$, ar presupune un automat pentru care

$$|Q| = 2n_0 + const. = O(n_0).$$

Cititorul poate verifica singur acest lucru. Dependenţa de lungimea şirului maxim de intrare face ca automatul rezultat să nu mai aparţină categoriei "automatelor finite" în sensul comentariului ce urmează *Definiţiei 5.4* în paragraful 5.4.1.

Soluţia oferită cu un *AS* este, însă, imediată, generînd o structură comod realizabilă. Organigrama ce descrie funcţionarea automatului finit asociat *AS* este reprezentată în figura 6.24. De cîte ori este recepţionat un *a*, acesta este introdus în stivă (*PUSH a*). Dacă urmează *b*-uri, pentru fiecare *b* recepţionat se comandă un *POP*, descărcîndu-se stiva. Dacă la terminarea *b*-urilor în stivă se găseşte $z_0$, atunci numărul de *a*-uri a fost identic cu cel al *b*-urilor şi automatul trece în starea *4*, semnalînd prin aceasta un şir bine format de tip $a^n b^n$, altfel reîncepe căutarea de la început. Acest automat are *5* stări indiferent de valoarea lui *n* care, de această dată, determină numai capacitatea minimă a stivei care trebuie să fie de $n+1$ locaţii.

Cazul particular al exemplului anterior putea fi rezolvat folosind în locul stivei numai un numărător reversibil. Complexitatea resursei dependente de *n* ar fi fost numai $O(\log n)$ în acest caz, faţă de $O(n)$ pentru memoria de tip stivă. *AS* posedă însă aplicaţii mai largi decît un automat conectat în buclă cu un numărător reversibil.

Aplicaţiile *AS* sînt legate, în primul rînd de teoria limbajelor formale.

## 6.4 Procesorul elementar

Structura reprezentativă pentru clasa sistemelor de ordinul trei o reprezintă *procesorul*. Sub diverse forme, pînă la sfîrşitul acestui capitol, ne vom ocupa de această categorie de circuite. Prin posibilităţile oferite în prelucrarea datelor, atunci cînd sînt folosite pentru structurarea sistemelor de calcul, procesoarele reprezintă componente ce înglobează astăzi toate eforturile de concepţie din domeniul sistemelor digitale, ele reprezintă un vîrf care odată atins, prefigurează şi o limită ce va trebui depăşită.

### 6.4.1 Principiul de structurare

**Definiţia 6.2** Definim drept *procesor* structura obţinută prin interconectarea unui *RALU* cu un automat iniţial de control, *AC*, conform reprezentării din figura 6.25. *AC* este definit prin



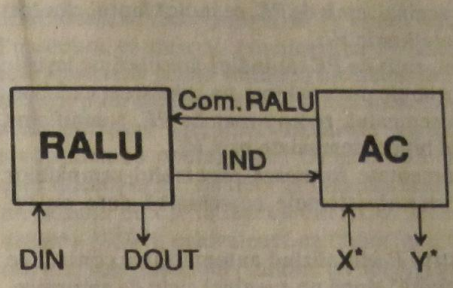



**Figura 6.25**

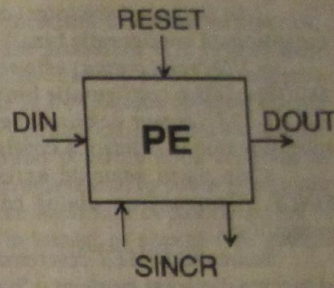



**Figura 6.26**

$$X = X^* \times IND$$
$$Y = Y^* \times Com.RALU,$$

unde *IND* şi *Com.RALU* sînt definite în paragraful 5.7.3 iar $X^*$ şi $Y^*$ reprezintă conexiuni din şi în afara procesorului. □

**Definiţia 6.3** Un *procesor elementar* (*PE*) este un procesor al cărui automat de control, *AC*, este *strict iniţial* şi evoluează numai sub comanda semnalelor *IND* şi $X^*$. □

Simbolul pe care-l putem utiliza pentru un *PE* este cel din figura 6.26 unde:

- intrarea *RESET* asigură iniţializarea sistemului (aducînd şi automatul de control în starea iniţială);

- conexiunile bidirecţionale *SINCR* asigură numai *sincronizarea fluxurilor de date* la bornele *PE* pe căile *DIN* şi *DOUT*.

**Exemplul 6.9** Pentru a exemplifica cel mai simplu procesor elementar vom folosi structura din figura 6.27 unde *RALU* a fost definit în paragraful 5.7.3 iar automatul *strict iniţial* format din *ROM/PLA*, *R* şi *MUXT* a rezultat prin simplificarea automatului *iniţial* din figura 5.84. Linia de comandă ce descrie funcţionarea sistemului va putea fi de această dată explicit definită după cum urmează:

<*Linie de comandă*> ::= <*Com. RALU*><*Out*><*TEST*><*NEXT*>,

<*Com. RALU*> ::= a fost definită în *Exemplul 5.33*,

<*Out*> ::= *IA* | *VO* | −,

<*TEST*> ::= *CR* | *OVFL* | *ZERO* | *EQUAL* | *PAR* | *SGN* | *VI* | *OA* | *UNU* | *NUL*,

<*NEXT*> ::= *0* | *1* | *10* | *11* | *100* | ... | *11...1*, (*n-1*)

unde:

- *VI* (*valid input*) este un semnal recepţionat ce indică faptul că la intrarea *PE* se află, pe căile *DIN*, o configuraţie binară ce trebuie recepţionată;

- *IA* (*input acknowledge*) este un semnal, emis de *PE*, ce indică faptul că a fost recepţionată configuraţia binară semnalată prin *VI*;

- *VO* (*valid output*) este semnalul, emis de *PE*, ce indică faptul că pe ieşirile *DOUT* se află o configuraţie binară ce trebuie preluată într-un subsistem exterior;

- *OA* (*output acknowledge*) este semnalul, recepţionat de *PE*, semnificînd recepţionarea în exterior a configuraţiei binare semnalate prin *VO*.

Cele patru semnale anterior comentate formează ansamblul semnalelor *SINCR* şi asigură un dialog coerent cu subsistemele ce schimbă date cu *PE* exemplificat.

Semnalul *RESET* resetează registrul *R* iniţializînd automatul de control ce va executa din acest moment o "buclă infinită", după un eventual ciclu de amorsare.

În fiecare ciclu de ceas se emite o configuraţie binară de comandă către *RALU* şi un semnal (semnificativ sau nu) pe calea *OUT*. Starea ulterioară este determinată de codul *NEXT* şi de bitul selectat prin cîmpul *TEST* prin care se configurează *ADR* ce selectează din *ROM/PLA* starea următoare (*NEXT, TEST*) şi comenzile următoare (*Com.RALU, OUT*). *ADR* se obţine conform relaţiei

$$ADR = 2 \times (NEXT) + T,$$

unde $\times$ şi $+$ sînt funcţii aritmetice binare. Altfel spus, *T* reprezintă bitul cel mai puţin semnificativ al *ADR* iar *NEXT*, pe ceilalţi. Tranziţiile necondiţionate, de indicatori sau de semnalele din exteriorul *PE*, se obţin prin valorile *NUL* şi *UNU* ale cîmpului *TEST*, cînd *T* ia valorile *0*, respectiv, *1*. □

*PE* fiind controlat de un automat strict iniţial va efectua o prelucrare de un singur tip a şirurilor de intrare recepţionate, emiţînd rezultate tot sub forma unui şir de configuraţii binare. Pentru a evidenţia modul de interacţie al *PE* cu subsisteme exterioare vom folosi exemplul simplu ce urmează.

**Exemplul 6.10** Vom considera aplicaţia simplă ce presupune recepţionarea unor şiruri de numere întregi şi pozitive, reprezentate pe *16* biţi şi emiterea mediei aritmetice realizată cu perechi de cîte două numere succesive. Deci, pentru fiecare

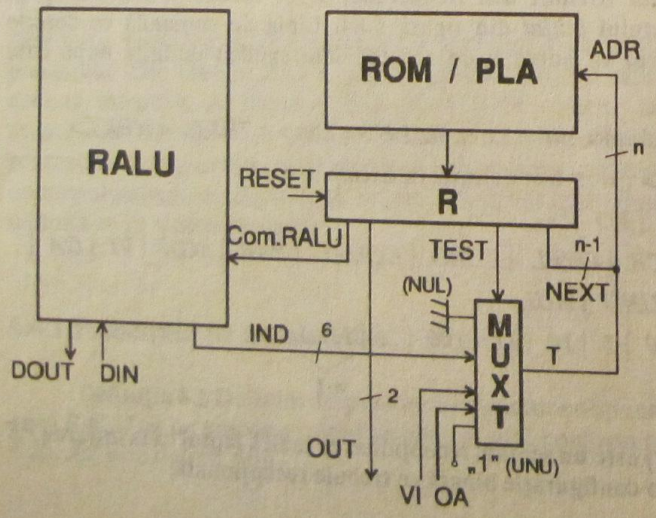



**Figura 6.27**



două numere recepţionate se va emite unul ce reprezintă media lor aritmetică. Procedura ce descrie funcţionarea *PE* este cea din procedura 6.1. Primul *loop* recepţionează prima variabilă încărcînd în *R0* (registrul zero din *RALU*) valoarea intrării pînă cînd apare *VI*. Ultima încărcare aduce în *R0* valoarea corectă. Primul *while* buclează "aşteptînd" dispariţia semnalului *VI* şi emiţînd *IA* pentru a anunţa recepţionarea operandului. Urmează un ciclu similar pentru a doua variabilă ce este stocată în *R1*. Se trece la sumarea celor doi întregi pozitivi ce poate genera sau nu semnal de *CR* la ieşirea din *ALU*. În funcţie de acest fapt după deplasarea la dreapta (*SHR*), echivalentă cu împărţirea cu doi în întregi, se adaugă *1* pe poziţia cea mai semnificativă (sumînd cu $8000_H$) sau nu.

```
Procedure MEDIE
    loop   loop R0 ← DIN  /R0 ia valoarea de la intrare./
                  until VI
           repeat
           while VI  / Se aşteaptă dispariţia semnalului VI./
                  do IA  / Se anunţă operand recepţionat./
           repeat
           loop R1 ← DIN
                  until VI
           repeat
           while VI
                  do IA
           repeat
           R0 ← R0 plus R1  / Sumare./
           if CR
                  then     SHR valoarea din R0  / Divizare./
                           R0 ← R0 plus 8000H
                  else     SHR valoarea din R0
           endif
           loop DOUT ← R0, VO  / Emite rezultatul./
                  until OA
           repeat
    repeat
end MEDIE
```

**Procedura 6.1**

Urmează emiterea rezultatului pînă cînd cel care-l recepţionează validează transferul prin *OA*. Bucla principală este necondiţionată, repetîndu-se "la infinit".

Transcrierea în linii de comandă înregistrate în *ROM/PLA* este următoarea (fiecare linie este numerotată în hexazecimal cu adresa la care este stocată):

$0_H$ *R0 LEFT W DIN 000(VI)*
$1_H$ *IA 001(VI)*
$2_H$ *R1 LEFT W DIN 010(VI)*
$3_H$ *IA 001(VI)*
$4_H$ *R1 LEFT W DIN 010(VI)*
$5_H$ *IA 011(VI)*
$6_H$ *L1 R0 PLUS W 100(CR)*
$7_H$ *IA 011(VI)*
$8_H$ *SHR W R0 L0 110(NUL)*
$9_H$ *SHR W R0 L0 101(NUL)*
$A_H$ *L0 R0 #8000 PLUS W 110(NUL)*
$B_H$
$C_H$ *L0 LEFT VO 110(OA)*
$D_H$ *000(NUL).*

Prima linie de comandă, $0_H$, în care automatul poate ajunge prin *RESET*, permite trecerea prin *ALU* (*LEFT*) a valorii binare de pe intrarea *DIN* şi înscrierea (*W*) în registrul zero (*R0*). *ROM*-ul (sau *PLA*-ul) este adresat prin *000(VI)*, pe poziţia cea mai puţin semnificativă fiind selectată prin *MUXT* valoarea bitului *VI*. Dacă, *VI=0*, atunci va fi selectată, în continuare linia de la adresa $0_H$ (adică *0000*). Dacă, *VI=1* (pe intrare se află un operand valid), atunci este selectat conţinutul locaţiei $1_H$ *(0001)* la care se află comanda *IA*, prin care se anunţă că operandul a fost recepţionat, şi se comandă salt la *001(VI)*, aşteptîndu-se, eventual, dispariţia semnalului *VI* în starea $3_H$. În $2_H$ se recepţionează cel de al doilea operand, în $6_H$ se realizează sumarea celor doi operanzi, stocaţi în registrele *0* şi *1*, rezultatul fiind depus la *R0*. Dacă nu se obţine depăşire (*CR*) la această operaţie, urmează împărţirea cu doi prin deplasare la dreapta (*SHR*) în starea $8_H$, altfel, în $9_H$ se face acelaşi lucru dar urmează adăugarea unui *1* pe poziţia cea mai semnificativă prin sumarea cu *#8000* în starea $A_H$. În starea $C_H$ se emite rezultatul iar din $D_H$ se revine în starea iniţială, reluîndu-se, astfel, bucla.

Aplicaţia presupune *12* linii de comandă stocate într-un *ROM* de *16* cuvinte. □

Dacă procedura 6.1 a fost foarte uşor de urmărit, trebuie să constatăm că secvenţa de comenzi din exemplul anterior se construieşte, se depanează şi se înţelege mult mai dificil. Dependenţa unui bit de adresare a *ROM*-ului de indicatorul testat îngreunează foarte mult scrierea secvenţei de comandă şi se pare că determină folosirea unui număr de linii de comandă sporit. De asemenea, faptul că în fiecare linie de comandă trebuia să apară explicit starea următoare



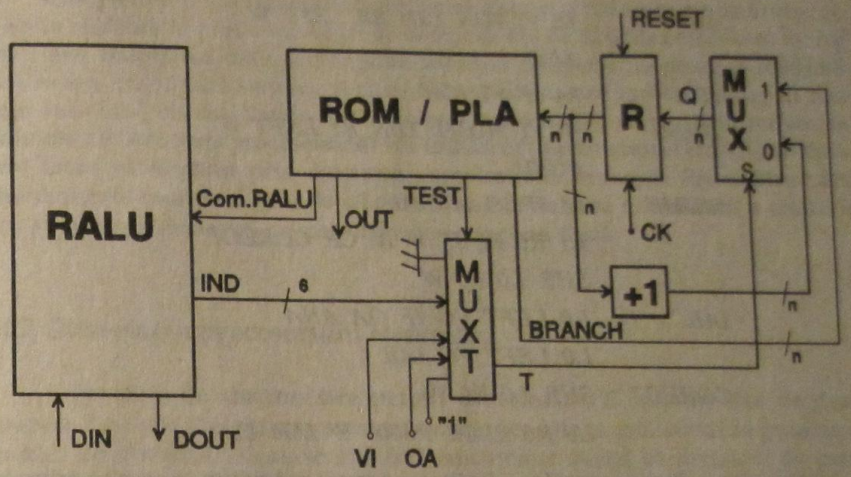



**Figura 6.28**

(următoarea adresă), complică scrierea unor secvenţe de comandă mai complexe. În exemplul următor vom da o variantă de *PE* mai prietenoasă faţă de utilizator.

**Exemplul 6.11** O variantă mai performantă de *PE* este reprezentată în figura 6.28, unde, pentru automatul de control, a fost utilizată varianta strict iniţială a *CROM*-ului din figura 5.87. Linia de comandă asociată acestui *PE* este următoarea:

*<Linie de comandă> ::= <Com. RALU><Out> IF <TEST><etichetă>*
*<TEST> ::= CR | OVFL | ZERO | EQUAL | PAR | SGN | VI | OA | –*
*<etichetă> ::= orice şir de maximum şase simboluri ce nu a fost folosit în definirea altor cîmpuri;*

iar celelalte cîmpuri au definiţia dată anterior. *IF* este o sintagmă ce indică saltul condiţionat.

Fiecare linie de comandă poate fi numită printr-o *etichetă*, referirea la ea făcîndu-se numai prin intermediul acesteia. Etichetarea, face numerotarea liniilor neobligatorie. Orice etichetă apărută în construirea unei linii de comandă trebuie să se regăsească în numele unei linii de comandă. Orice etichetă nu poate denumi decît cel mult o singură linie de comandă. Pe scurt: etichetele trebuie să fie *definite* şi să fie *unice*.

Dacă, o linie de comandă nu conţine nici o etichetă, atunci, este urmată de linia de comandă imediat următoare (tranziţia prin incrementarea stării cu structura *+1*). Dacă apare şirul *IF <TEST> <etichetă>*, tranziţia este condiţionată la *etichetă* în sensul că dacă este îndeplinită condiţia se realizează saltul, altfel se trece la linia imediat următoare. Dacă, apare numai eticheta, atunci, tranziţia este necondiţionată, la linia marcată prin ea. □

**Exemplul 6.12** Să reluăm *Exemplul 6.10* cu *PE* anterior definit. Va rezulta următoarea secvenţă de comenzi:

```
ANA     IF VI ION DIN R0 LEFT W
        ANA
ION     IA IF VI ION
BUBU    IF VI MUMU DIN R1 LEFT W
        BUBU
MUMU    IA IF VI MUMU
        L1 R0 PLUS W IF CR CORECT
        SHR L0 R0 W
IAR     L0 LEFT VO IF OA ANA
        L0 LEFT VO IAR
CORECT  SHR L0 R0 W
        L0 R0 PLUS #8000 W IAR □
```

Nu putem să nu remarcăm simplitatea acestei noi soluţii în ceea ce priveşte, în primul rînd, o translaţie comodă de la pseudocod la secvenţa de comenzi. Numărul de linii nu s-a redus spectaculos, dar acest lucru este posibil printr-o uşoară modificare a maşinii. Singura dificultate resimţită a fost aceea că nu am putut găsi o realizare la fel de simplă ciclului *loop* şi ciclului *while*, deoarece nu avem posibilitatea să testăm şi un indicator şi negatul lui. Vom corecta în următorul exemplu această deficienţă.

**Exemplul 6.13** Definim un *PE* a cărui linie de comandă este de următoarea formă:

*<Linie de comandă> ::= <Com. RALU><Out>IF NOT<TEST><etichetă>.*

Cîmpurile ce apar au fost anterior definite.

Posibilitatea apariţiei sintagmei *NOT* permite să ţinem cont, la o tranziţie condiţionată a automatului de control, şi de valoarea negată a unui bit testat.

Structura fizică ce corespunde acestei definiţii va fi imaginată de cititor. □

**Exemplul 6.14** Reluînd aceeaşi problemă cu ultima definiţie a *PE* rezultă:

```
* ANA     DIN R0 LEFT W IF NOT VI ANA
  ION     IA IF VI ION
  BUBU    DIN R1 LEFT W IF NOT VI BUBU
  MUMU    IA IF VI MUMU
          L1 R0 PLUS W IF CR CORECT
          SHR L0 R0 W
  IAR     L0 LEFT VO IF NOT OA IAR
          ANA
  CORECT  SHR L0 R0 W
          L0 R0 PLUS #8000 W IAR □
```



Completînd puţin structura fizică am obţinut şi o reducere a numărului de linii şi o claritate în plus a secvenţei de comenzi. Cel de al doilea efect este foarte important. Stăpînirea unor secvenţe de comenzi complexe nu poate fi realizată decît în măsura în care scrierea şi citirea lor se poate face într-un limbaj cît mai puţin restrictiv, cît mai "prietenos". Multe din facilităţile ce vor fi introduse în continuare în structura procesoarelor vor urmări obţinerea acestui efect, chiar dacă acest lucru va fi plătit prin creşterea complexităţii structurii fizice. Structura simbolică este mult mai greu de stăpînit deoarece va avea o dinamică a creşterii mult superioară dinamicii cu care va evolua structura fizică.

### 6.4.2 Conectarea procesorului elementar

Semnalele de sincronizare permit conectarea *PE* într-un flux de date asincron. Testarea şi generarea acestor semnale permite ca preluarea sau generarea şirurilor de simboluri să nu se mai facă sincronizat numai cu semnalul de ceas. Apare un mecanism suplimentar de sincronizare care permite o ritmicitate flexibilă, impusă de mecanisme de prelucrare ce presupun algoritmi ce au complexităţi diferite. Mai mult, complexitatea prelucrării nu depinde întotdeauna numai de *tipul prelucrării* ci şi de *datele prelucrate*. Problema rezolvată, prin mai multe procedee, în paragraful anterior este un bun exemplu în acest sens. Într-adevăr, calculul mediei se efectua într-un număr de paşi (de stări) ce depindea şi de numerele mediate. Dacă, numerele erau suficient de mari ca sumarea să se realizeze cu depăşire (*CR*), atunci, media respectivă era calculată într-un interval de timp ce presupunea un ciclu de ceas în plus. Astfel de abateri de la o ritmicitate rigidă sînt posibile numai datorită cuplării flexibile prin semnalele de sincronizare.

- **Conectarea PE pe fluxul de date transferate între două sisteme** constituie una din aplicaţiile curente ale acestei structuri. În figura 6.29, *S1* şi *S2* sînt sisteme oarecare, unul generează date către *PE* iar celălalt recepţionează rezultatele emise în urma procesării. Un exemplu posibil ar fi acela al realizării unei structuri ce conţine un *filtru digital* realizat cu o structură de *PE*. Funcţia, unică, controlată de automatul din *PE* ar fi cea de realizare a funcţiei de transfer asociate unui filtru digital care recepţionează şiruri de eşantioane dintr-un semnal şi emite şiruri de numere ce constituie eşantioanele semnalului filtrat.

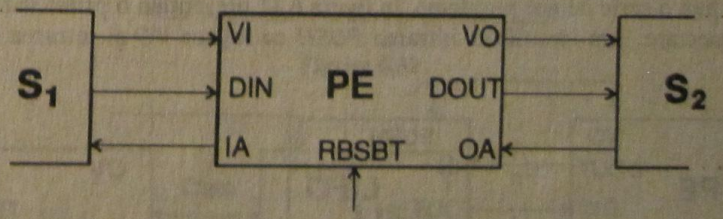



**Figura 6.29**

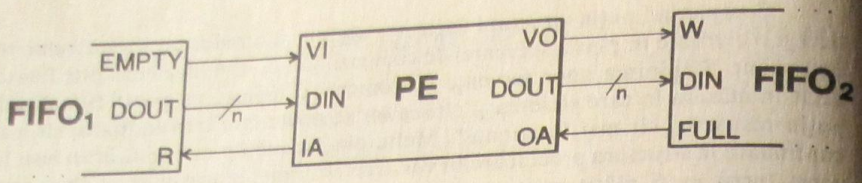



**Figura 6.30**

Un al doilea exemplu posibil ar putea fi realizarea cu *PE* a unui *procesor Fourier* ce calculează, pe baza *transformatei Fourier discrete* (sau a unor algoritmi îmbunătăţiţi), pornind de la eşantioanele *în timp* ale unui semnal, eşantioanele *în frecvenţă* ale acestuia. Printr-o dimensionare convenabilă a numărului de registre se poate calcula transformata Fourier pe un număr suficient de mare de eşantioane. Aplicaţiile cele mai spectaculoase ale *PE* sînt legate, de regulă, de *procesarea digitală a semnalelor*.

• **Conectarea PE prin FIFO** permite realizarea unor structuri în care fluxul de date să fie mai puţin restricţionat, să se realizeze mai fluent. Datorită faptului că procesarea depinde şi de valorile concrete, ce sînt percepute ca mărimi de intrare, ritmicitatea la bornele unui *PE* nu poate fi garantată întotdeauna cu rigurozitatea unui semnal de ceas. Pentru ca semnalele de dialog să restricţioneze cît mai puţin transferul datelor, sînt uneori folosite, la bornele *PE*, memorii de tip *FIFO*, conform reprezentării din figura 6.30.

În majoritatea cazurilor *FIFO 1* nu este golită (*EMPTY = 0*) iar *FIFO 2* nu este plină (*FULL = 0*), permiţînd recepţionarea sau emiterea unei configuraţii binare. Conform definiţiei din subcapitolul 4.7 şi figurii 4.41 am identificat pe *EMPTY* cu *VI* şi *FULL* cu *OA* iar pe *R* cu *IA* şi *W* cu *VO*. În aceste condiţii liniile de comandă asociate comunicării *PE* vor fi de forma

*NOI* *DIN R0 LEFT IF VI NOI*

pentru recepţie şi

*ION* *IF OA ION*
*L0 LEFT VO*

pentru emisie în şi din registrul *R0*.

• **Conectarea PE la un LIFO** poate fi cerută în anumite aplicaţii, ocazie cu care se pun o serie de noi probleme. În figura 6.31 prezentăm o primă variantă de interconectare. Am identificat intrarea *PUSH* cu ieşirea *VO* şi intrarea *POP* cu

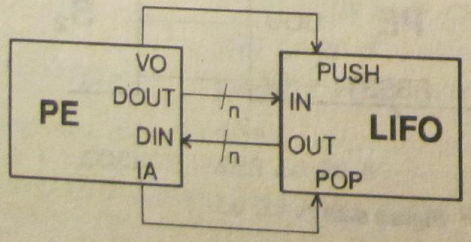



**Figura 6.31**



ieşirea *IA*, fapt permis de modul în care funcţionează o memorie de tip *LIFO* (conform paragrafului 5.8.2). Liniile de comandă asociate comunicării cu memoria stivă vor fi de următorul tip:

*DIN R0 LEFT IA*

pentru citirea în *R0* şi

*L0 LEFT VA*

pentru scrierea din acelaşi registru. De regulă aceste memorii sînt realizate în tehnologii rapide putînd răspunde prompt (în cadrul unui singur ciclu de ceas) comenzilor primite. În aplicaţiile curente capacitatea acestor memorii este gîndită ca fiind suficient de mare ("infinită"). De asemenea, numărul de *POP*-uri nu-l poate depăşi pe cel de *PUSH*-uri într-o aplicaţie corect gîndită. Aceste ultime două observaţii fac inutil dialogul pe căile de sincronizare logică, permiţînd comenzi imperative de comunicare a *PE* cu *LIFO*.

• **Conectarea bidirecţională pe căile de date** este sugerată, eventual impusă, de faptul că *LIFO* este o structură compactă iar căile de date presupun multe conexiuni (*16, 32* sau mai multe). Minimizarea numărului de conexiuni se poate realiza conectînd *PE* cu *LIFO* pe un *bus bidirecţional*, folosind configuraţia de circuit din figura 6.32. Circuitele *tristate* permit transferul bidirecţional între două subsisteme. Inversoarele standard folosite refac semnalul inversat de driverele tristate. De asemenea, aceleaşi circuite asigură o recepţionare corectă a semnalului electric în situaţia unei conexiuni realizată pe trasee ce se comportă ca linii lungi.

Circuitele ce asigură transferul bidirecţional putînd fi înglobate în structurile pe care le interconectează, rezultă reprezentarea din figura 6.33. Aceste circuite de interconectare, pe lîngă funcţia logică pe care o au, asigură şi o interfaţare electrică adecvată.

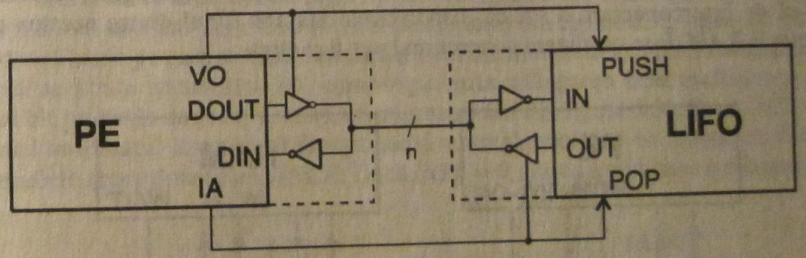



**Figura 6.32**

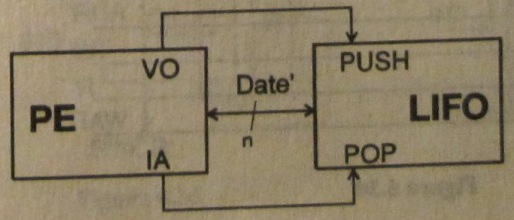



**Figura 6.33**

• **Conectarea PE la o memorie RAM** este pasul următor ce poate fi făcut, cu principala implicaţie dată de faptul că se impune şi o conexiune pe care să se transfere *adresa* locaţiei de memorie. Memoriile de tip *FIFO* sau *LIFO* presupun numai comenzi de scriere şi citire deoarece au o structură bine precizată (şir cu acces la ambele capete sau numai la un capăt). Memoria *RAM* nefiind structurată, accesul se face prin indicarea explicită a adresei. Structura internă a *PE* se va modifica astfel încît la borne să fie disponibilă configuraţia binară ce indică adresa. (O soluţie posibilă este ca adresa să fie generată pe ieşirea *LOUT* din *RALU* ce a fost reprezentat în figura 5.97.)

Modul de conectare cu un *RAM* este reprezentat în figura 6.34, unde se individualizează următoarele conexiuni:

- căile bidirecţionale de date pe care semnalele circulă negate (*DATE'*), conform principiului sugerat în figura 6.32;

- căile de adrese (*ADR'*), de asemenea active pe zero;

- semnalul de scriere *W'* (write), activ pe zero;

- semnalul de citire *R'* (read);

- semnalul *WAIT'* care asigură sincronizarea logică împreună cu cele două anterioare; acest semnal este activ (pe zero) atît timp cît este nevoie ca biţii de adresă, eventual şi datele, să fie stabili pe conexiunile destinate, pentru ca transferul (scriere sau citire) să se efectueze corect.

În cazul unor memorii *RAM* realizate în tehnologii suficient de rapide semnalul *WAIT'* nu se activează niciodată. Dacă, timpul de acces la memorie este mai mare decît cel permis de transferul într-un ciclu al procesorului, atunci acest semnal se va activa un interval de timp suficient pentru citirea sau scrierea comandată.

Grupul de conexiuni anterior descris defineşte ceea ce, de regulă, reprezintă *bus*-ul de interconectare a *PE* cu dispozitivele ataşate. Unul dintre acestea este memoria *RAM*. Într-un sistem concret mai pot fi şi altele.

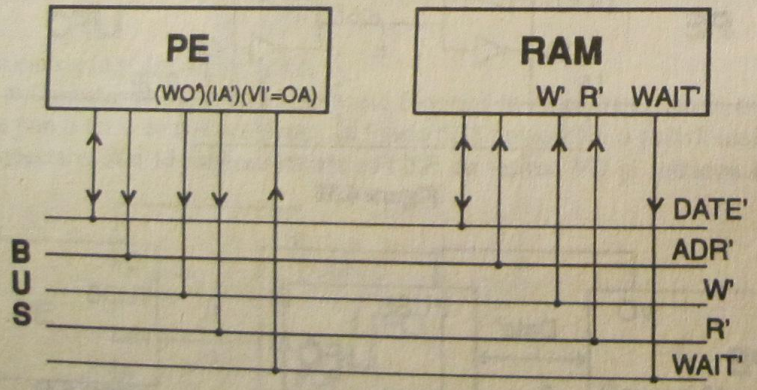



**Figura 6.34**



• **PE este o structură orientată funcţional** în sensul că poate fi astfel structurat încît să realizeze *o funcţie şi numai una*, căreia îi este strict dedicat şi pentru care este optimizat. Se pot imagina variante structurale diferite de cele expuse în această secţiune, dar caracteristica anterior enunţată este păstrată. Pentru clase destul de largi de probleme se pot folosi structuri asemănătoare, distinse numai prin automatul de control, numărul de registre din *RALU* sau prin funcţiile unităţii *ALU*.

Caracterisică pentru *PE* este existenţa unui *automat efector* (spre exemplu *RALU*) şi a unui *automat de control* (spre exemplu *CROM*) realizat ca un automat *strict iniţial*.

• **Ce se întîmplă cînd automatul de control nu mai este strict iniţial?** Posibilitatea iniţializării în mai multe stări permite declanşarea mai multor secvenţe distincte de control asociate unor funcţii distincte, aplicate şirurilor de simboluri recepţionate sau configuraţiilor binare stocate în registrele interne.

Un automat de control iniţial are asociat un *set de funcţiuni* elementare şi nu numai o singură funcţie. Aceste funcţii pot fi declanşate din exteriorul structurii. Vom defini explicit, pe această cale, conceptul de *procesor*.

## 6.5 Procesorul

În cazul general, procesorul a fost anterior definit (*Definiţia 6.2*) pentru a fi imediat particularizat sub forma procesorului elementar (*PE*). Putem, acum, reveni la definiţia iniţială în tentativa de a specifica structural ceea ce este un procesor.

Pornind de la structura reprezentată în figura 6.28, vom reţine într-o formă simplificată automatul de control sub forma celui din figura 6.35, unde am pus în evidenţă bucla pe care se transferă elementele din mulţimea *Q*. Acest automat *strict iniţial* se poate transforma într-unul *iniţial* prin adăugarea unui multiplexor şi a unui bit pe bucla de reacţie, conform reprezentării din figura 6.36. Dacă *INIT = 1*, atunci automatul de control funcţionează normal, evoluînd în funcţie de *IND* şi semnalele recepţionate pe *SINCR*. Dacă *INIT = 0*, starea următoare a automatului

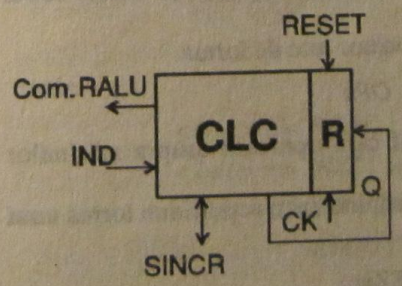



**Figura 6.35**

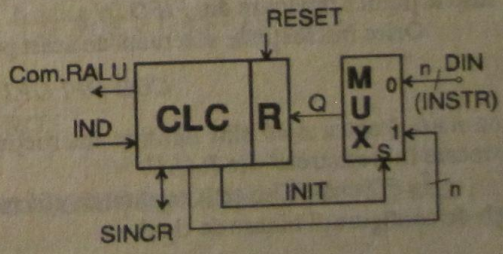



**Figura 6.36**

este dată de configurația binară ce vine pe căile *DIN*, din exteriorul procesorului. În acest caz *automatul se inițializează* într-o stare dictată de o configurație binară recepționată pe căile de date. Procesul de inițializare se află sub controlul automatului, dar starea în care comută automatul în acest caz este determinată de un proces extern.

Folosind un astfel de *automat inițial* într-o configurație de tipul celei din figura 6.25, se realizează o structură de procesor minimală. Vom distinge două variante:

- procesoare cuplate cu memorii ce nu presupun adresare, ci numai comenzi (ex: *LIFO*, *FIFO*), reprezentate în figurile 6.37 și 6.38;

- procesor cuplat cu memorie *RAM*, reprezentat în figura 6.41.

### 6.5.1 Procesoare fără bus de adresă

O primă categorie de procesoare o constituie cele ce-și extrag operanzii și funcțiunile și depun rezultate finale sau parțiale în și din memorii la care accesul nu presupune o adresă explicită. Este vorba, spre exemplu, de memorii ce stochează structuri de date sub formă de șiruri cu acces la ambele capete (*FIFO*) sau numai la un capăt (*LIFO*).

• **Procesorul fără bus de adresă, într-o primă variantă,** este reprezentat în figura 6.37 funcționînd cu două memorii *FIFO*. Automatul de control (*AC*) comandă citire (*R*) din *FIFO IN* și scriere (*W*) în *FIFO OUT*, recepționînd semnalele de *FULL* și *EMPTY* ca pe doi indicatori. Structura datelor în *FIFO IN* se prezintă sub forma unui șir de tipul următor:

*...CODi OP1 OP2 ... OPn CODj OP1 OP2 ... OPm CODk OP1 ...*

unde *COD i, j* sau *k* reprezintă codul instrucțiunii de executat (adresa de inițializare a *AC*), iar *OP1*, *OP2*, ... reprezintă operanzii asociați instrucțiunii ce precedă subșirul operanzilor. *CODi*, spre exemplu, va fi recepționat în *AC* pe calea *INSTR*, iar *OPj* pe calea *DIN* în *RALU*.

Secvența de comenzi declanșată de *CODi* "știe" cîți operanzi trebuie recepționați pentru fiecare instrucțiune. Odată operanzii recepționați și instrucțiunea efectuată, se emit sau nu rezultate spre *FIFO OUT*, se citește codul instrucțiunii următoare din *FIFO IN* ș.a.m.d.

Orice instrucțiune efectuată de acest procesor este de forma

*COD OP1 OP2 ... OPn*

cu $n \geq 0$. Pentru $n=0$ sînt definite instrucțiuni care operează asupra mărimilor stocate în "registrele" din *RALU*.

În *FIFO OUT* se emit rezultatele efectuării unei instrucțiuni sub forma unui șir de configurații binare de tipul

*REZ1 REZ2 ... REZm,*



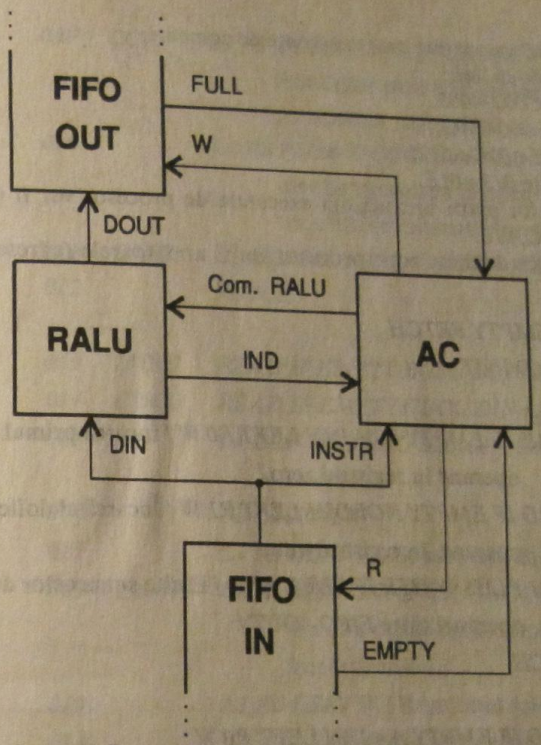



**Figura 6.37**

unde $m \geq 0$. Pentru $m=0$ rezultatul rămîne stocat în "registrele" din *RALU* pentru a fi prelucrat ulterior, eventual de o instrucțiune pentru care $n=0$.

Un caz limită este acela în care rezultatele sînt complet emise și nu există instrucțiune care să presupună găsirea unui operand în memoria *RALU*. În acest caz $n \neq 0$ și $m \neq 0$ și rezultatul nici unei instrucțiuni nu depinde de cel al instrucțiunilor anterior executate.

**Exemplul 6.15** Vom exemplifica cu un procesor aritmetic, foarte simplu, ce execută cele patru operații aritmetice modulo $2^{16}$ cu întregi pozitivi. Considerăm un *AC* (figura 6.36) cu următoarea linie de comandă

*<Linie de comandă> ::= <Com. RALU><Out><Init>*
*IF NOT <Test><Etichetă>.*

unde:

*<Out> ::= READ | WRITE | –*
*<Init> ::= INIT | –*
*<Test> ::= CR | OVFL | ZERO | EQUAL | PAR | SGN | FULL |*
*EMPTY | –*

iar celelalte cîmpuri anterior definite.

Memoria *ROM* a *AC* va conţine patru secvenţe de comenzi:
- *FETCH*, primele două linii;
- *ADUNARE*, liniile *002÷005*;
- *SCĂDERE*, liniile *006÷00B*;
- *ÎNMULŢIRE*, liniile *00C÷012*;
- *ÎMPĂRŢIRE*, liniile *013÷01E*.

În consecinţă, codurile celor patru instrucţiuni executate de procesor vor fi (în hexazecimal): *002*, *006*, *00C*, *013*.

Liniile de comandă ce definesc acest procesor vor fi următoarele (adresele sînt cele fizice):

```
000   FETCH  IF EMPTY FETCH
001          READ INIT
------------
002   ION    READ IF EMPTY ION DIN LEFT R0 W / Încarcă primul
               operand în registrul zero./
003   NOI    READ IF EMPTY NOI DIN LEFT R1 W / Încarcă al doilea
               operand în registrul unu./
004   ANA    L0 R1 PLUS WRITE IF FULL ANA / Emite suma celor doi
               operanzi către FIFO OUT./
005          FETCH
------------
006   AA     READ IF EMPTY AA DIN LEFT R0 W
007   BB     READ IF EMPTY BB DIN LEFT R1 W
008          L0 R1 MINUS W IF CR CC / Se scad cei doi operanzi, rezul-
               tatul este stocat în R1 şi se testează dacă operandul
               secund a fost mai mare decît primul./
009   DD     L1 LEFT WRITE IF FULL DD / Se emite rezultatul din re-
               gis trul unu./
00A          FETCH
00B   CC     R1 #0 RIGHT W DD / Rezultatul se forţează la valoarea ze-
               ro pentru că lucrăm cu o reprezentare ce nu admite
               numere negative./
------------
00C   AAA    READ IF EMPTY AAA DIN LEFT R0 W
00D   BBB    READ IF EMPTY BBB DIN LEFT R1 W
00E          L2 R2 XOR W / Se şterge conţinutul registrului doi în care
               se va forma rezultatul./
```



```
00F   CCC    L1 R1 #1 MINUS W IF CR EMIT / Se scade o unitate din
               înmulţitor pînă cînd devine zero semnalîndu-se astfel
               terminarea operaţiei./
010          L0 R2 PLUS W CCC / Se adună deînmulţitul în registrul în
               care se formează rezultatul pentru că înmulţitorul
               nu a fost "consumat" prin decrementare./
011   EMIT   L2 LEFT IF FULL EMIT WRITE
012          FETCH
------------
013   BUBU   READ IF EMPTY BUBU DIN LEFT R0 W
014   CUCU   READ IF EMPTY CUCU DIN LEFT R1 W
015          L1 LEFT IF ZERO ERROR / Se testează dacă împărţitorul
               este nul./
016          R2 L2 XOR W
017          L1 R3 LEFT W / Se salvează împărţitorul./
018   LOOP   L0 R1 MINUS W IF CR GATA / Se scade din deîmpărţit, sau
               din ce a rămas din el după mai multe cicluri, valoarea
               împărţitorului./
019          L1 R0 LEFT W / Rezultatul scăderii în R0./
01A          L3 R1 LEFT W / Se restaurează împărţitorul./
01B          L2 R2 #1 PLUS W LOOP / De cîte ori împărţitorul se mai
               cuprinde în deîmpărţit se imcrementează conţinutul
               registrului doi, unde se formează rezultatul./
01C   GATA   WRITE IF FULL GATA L2 LEFT
01D          FETCH
01E   ERROR  R2 #FFFF RIGHT W GATA / Se încarcă în registrul doi "in-
               finitul" reprezentării./
```

Fără a avea pretenţia unei maşini performante putem spune că acest procesor, definit prin *31* linii de comandă numai, rezolvă, într-o reprezentare rudimentară, cele patru operaţii aritmetice. Performanţele algoritmilor sînt modeste, dar scopul urmărit a fost numai acela de a ilustra ideea cu un exemplu dus pînă la capăt. Cititorul poate calcula timpii medii şi maximi asociaţi efectuării fiecărei operaţii. De asemenea, cititorul poate constata faptul că setul de funcţii ale *ALU* ar putea fi îmbunătăţit pentru a se rezolva mai bine funcţia de împărţire. □

**Exemplul 6.16** Să presupunem că în *FIFO IN* asociat procesorului din exemplul anterior se află următoarea secvenţă de configuraţii binare:

*xxxx xx00 0000 0110*
*1000 0000 0110 0000*
*0001 0011 0000 0000*
*xxxx xx00 0000 0010*
*1100 0001 0100 1000*
*1111 0000 0000 1010*
*xxxx xx00 0000 0110*
*0000 1000 1100 1100*
*0001 0000 0011 0001*
*xxxx xx00 0000 1100*
*0000 0000 0001 0000*
*0000 0000 1000 0000*
*xxxx xx00 0000 1100*
*1000 0000 0000 0000*
*0000 0000 0000 0011*
*xxxx xx00 0001 0011*
*0000 1000 0001 1111*
*0000 1100 0010 0001*
*xxxx xx00 0001 0011*
*0000 0000 0001 0000*
*0000 0000 0000 0011.*

Către *FIFO OUT* vor fi emise următoarele configuraţii binare drept rezultat al procesării:

*0110 1101 0110 0000*
*1011 0001 0100 1010*
*0000 0000 0000 0000*
*0000 1000 0000 0000*
*1000 0000 0000 0000*
*0000 0000 0000 0000*
*0000 0000 0000 0101.*

La a treia operaţie, scădere, rezultatul este zero pentru că al doilea operand este mai mare decît primul iar reprezentarea folosită nu admite numere negative. La a cincea operaţie, una de înmulţire, rezultatul a fost calculat modulo $2^{16}$, deci, $(3\times 2^{15})_{mod 2^{16}} = 2^{15}$. Rezultatul primei împărţiri este zero pentru că reprezentarea nu admite decît întregi. □



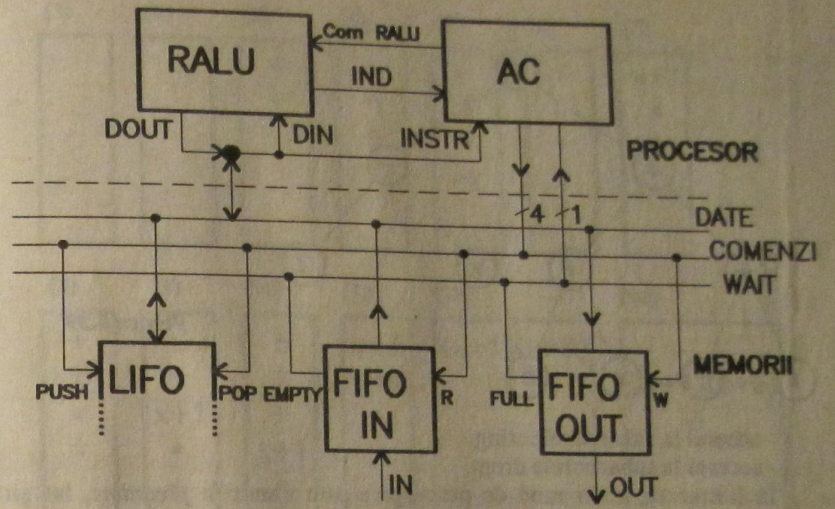



**Figura 6.38**

Limita esenţială a acestui tip de procesor (Exemplul 6.15) o constituie faptul că nu pot fi evaluate, într-o manieră elegantă, expresii. Cu un astfel de procesor se pot efectua numai cele patru operaţii. Completarea setului de instrucţiuni cu unele ce transferă conţinutul unor registre în altele şi cu operaţii definite asupra conţinutului unor registre, ar permite şi evaluarea unor expresii. O variantă mai flexibilă pentru astfel de aplicaţii o reprezintă următorul tip de procesor.

● **Procesorul orientat pe stivă** este reprezentat în figura 6.38 unde datele se transferă pe un *BUS* bidirecţional. Comenzilor externe pe care le avea procesorul anterior (figura 6.37) li se adaugă cele către o memorie *LIFO*, la care datele se conectează bidirecţional. Memoria *LIFO* este folosită pentru stocarea temporară a unor configuraţii binare, oferind posibilitatea evaluării unor expresii oricît de complexe.

**Propoziţia 6.1** Oricărei expresii formate din funcţii $n$-are i se poate asocia un arbore $n$-ar. □

**Exemplul 6.17** Să considerăm expresia ($n=2$)

$$f(x) = (x+1)(x+5)/(x-2).$$

Arborele binar ($n=2$) asociat acestei expresii este reprezentat în figura 6.39. Frunzele acestui arbore conţin operanzi: variabile ($x$) sau valori ($1$, $5$ şi $2$). În rest nodurile conţin funcţii cu operanzi ca succesor stîng şi succesor drept. Astfel operaţia minus (-) are ca operanzi pe $x$ şi $2$; înmulţirea operează asupra rezultatului celor două sumări iar împărţirea asupra rezultatului înmulţirii şi rezultatului scăderii. Ultima operaţie se află în rădăcina arborelui. □

Există mai multe reguli de parcurgere a unui arbore astfel încît să rezulte un şir. Ne vom opri la cea care presupune

- accesul la rădăcină,

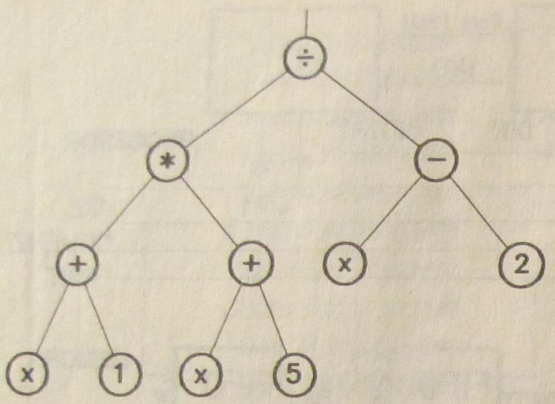



Figura 6.39

- accesul la subarborele stîng,
- accesul la subarborele drept.

În literatură, acest mod de parcurgere este numit *în preordine*, iar şirul rezultat conţine expresii în *forma prefixată*.

**Exemplul 6.18** Arborele din exemplul anterior poate fi parcurs, conform modalităţii expuse, după cum urmează:

$$\div * + x\,1 + x\,5 - x\,2$$

rezultînd un şir ce poate fi uşor "înţeles" şi evaluat de un procesor orientat pe stivă. □

**Exemplul 6.19** Dacă în *FIFO IN* se află forma prefixată a expresiei *f(x)* din Exemplul 6.17, atunci procesorul preia şirul simbol cu simbol şi-l transferă în *LIFO* după o regulă ce este ilustrată în figura 6.40, unde este reprezentat conţinutul stivei în etapele principale ale evaluării. Tot în *LIFO* sînt stocate şi rezultatele parţiale. Imediat ce în stivă apar doi operanzi, se realizează operaţia binară ce le este asociată. Spre exemplu, în etapa (7) se descarcă stiva de *1*, *x* şi +, se efectuează în procesor adunarea şi se reintroduce în stivă valoarea *x+1*. La fel în etapele (11) şi (12). Pe măsură ce şirul este recepţionat din *FIFO IN* se efectuează operaţiile pentru care operanzii sînt disponibili drept urmare a recepţionării lor (spre exemplu în (7)) sau drept urmare a unui calcul (spre exemplu în (12)). În final valoarea *f(x)* este extrasă din *LIFO* şi emisă către *FIFO OUT*.

Nici o operaţie nu lasă "urme" în procesor, acestea sînt construite ordonat în *LIFO*. □

Nu pentru orice fel de expresie o astfel de abordare este eficientă!

**Exemplul 6.20** Să se scrie în forma prefixată expresia

$$f(x) = (x+1)^5$$

pentru un procesor ce execută numai cele patru operaţii aritmetice elementare. Rezultă şirul:

$$* * * * + x\,1 + x\,1 + x\,1 + x\,1 + x\,1.$$



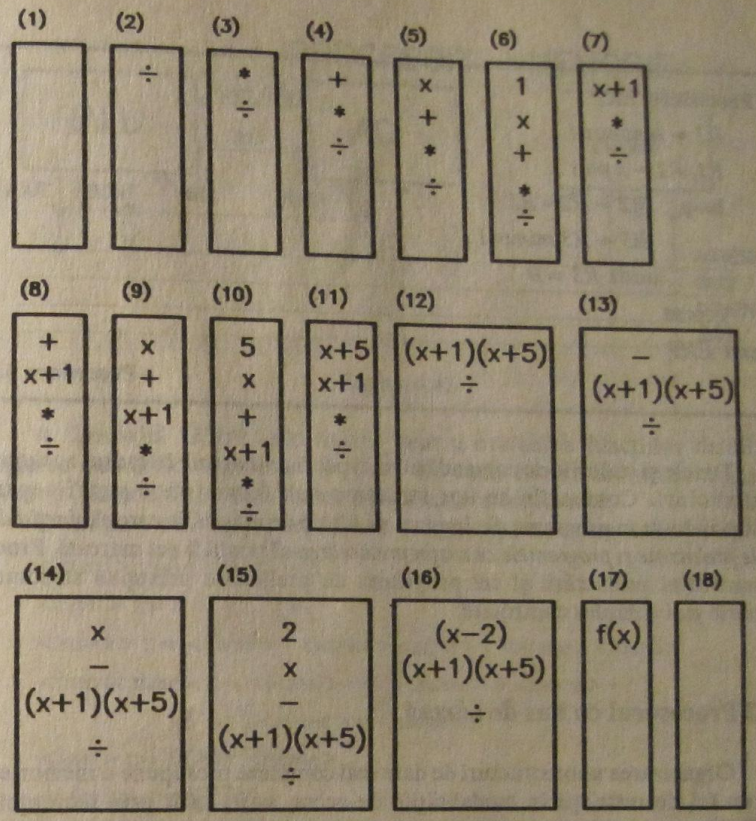



Figura 6.40

Ineficienţa evaluării într-o astfel de manieră este evidentă, impunîndu-se o altă configuraţie de procesare. □

Pentru exemplul anterior ar fi interesantă o manieră de evaluare descrisă prin procedura 6.2, pentru $n \geq 2$. Pe lîngă operaţiile utile:

*R1, R2 ← x plus 1,*

*R2 ← R2 * R1;*

apar şi unele parazite legate de controlul evaluării:

*R3 ← n minus 1,*

*R3 ← R3 minus 1,*

şi testul

*R3 = 0.*

```
Procedure EXP
    R3 ← n minus 1
    R1, R2 ← x plus 1
    loop    R2 ← R2 * R1
            R3 ← R3 minus 1
        until R3 = 0
    repeat
end EXP
```

**Procedura 6.2**

Datele şi codurile de comandă nu mai pot fi *amestecate* în cadrul aceluiaşi şir de simboluri. Comenzile se vor structura sub forma unor şiruri separate, constituindu-se în programe. Se impune un alt tip de maşină în care *distincţia dintre datele prelucrate şi programele ce susţin prelucrarea* să poată fi net marcată. Procesul de control al prelucrării şi cel propriuzis de prelucrare presupun structuri de memorie mai complex controlate.

### 6.5.2 Procesorul cu bus de adresă

Organizarea unor structuri de date mai complexe presupune o memorie fără nici un fel de restricţii în modalităţile de acces, astfel încît prin secvenţele de comandă generate de procesor (mai exact ale *AC*) să se poată structura diverse tipuri de funcţii de memorie. Memoria *RAM* este cea care oferă posibilităţi maxime în acest sens datorită faptului că nu posedă nici un fel de structură predeterminată. (Flexibilitatea maximă nu garantează, însă, şi performanţa maximă. Deci, atenţie!)

• **Procesorul cuplat cu memorie RAM** este reprezentat în figura 6.41. Tipul de memorie folosit impune şi o cale, suplimentară, de adresă, obţinută la ieşirea *LDOUT* din *RALU* (figura 5.97). Conţinutul oricărui registru din *RALU* poate deveni adresă pentru memoria *RAM*. Pentru a scrie în *RAM* conţinutul registrului *R5* la adresa conţinută în *RB* vom scrie linia de comandă următoare:

*ABA WRITE IF WAIT ABA R5 RIGHT LB*

iar pentru a citi în *R2* conţinutul locaţiei de memorie indicată de valoarea din *R9* vom scrie:

*BAB READ IF WAIT BAB DIN LEFT R2 L9.*

Un astfel de procesor poate fi asociat unui limbaj cu caracter general ce evaluează funcţii numerice. Vom considera, în continuare, unul suficient de simplu pentru a putea fi exemplificat prin particularizarea unor structuri deja studiate.



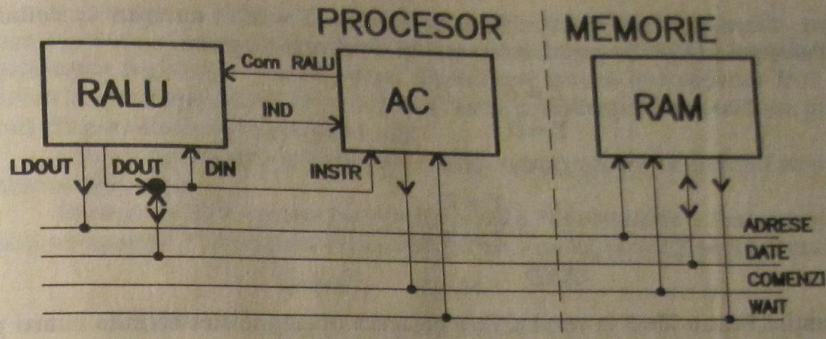



**Figura 6.41**

• **Limbajul LOOP** este folosit pentru evaluarea funcţiilor definite pe mulţimea numerelor naturale ($f$: $N^k \to N$, unde N este mulţimea numerelor naturale) [Calude '82]. Acest limbaj este descris în continuare.

**Definiţia 6.4** Limbajul *LOOP* este definit după cum urmează:

<*literă*> ::= *A* | *B* | ... | *Z* ,

<*cifră*> ::= *0* | *1* | ... | *9*,

<*nume*> ::= <*literă*> | <*nume*><*cifră*> | <*nume*><*literă*>,

<*instrucţiune*> ::= <*nume*>=*0* | <*nume*>=<*nume*>+*1* |
| <*nume*>=<*nume*>,

<*lanţ*> ::= *LOOP* <*nume*>,

<*sfîrşit*> ::= *END*,

<*program*> ::= <*lanţ*><*program*><*sfîrşit*> | <*program*><*program*> |
| <*instrucţiune*>,

unde secvenţa

```
LOOP    X
    P
END
```

semnifică faptul că programul *P* se execută de *X* ori. □

**Exemplul 6.21** În limbajul *LOOP* adunarea a două numere ($Z = X+Y$), se realizează cu următorul program:

```
Z = Y
LOOP  X
    Z = Z + 1
END
```

Observăm că valoarea lui *Y* este incrementată de *X* ori. □

**Exemplul 6.22** Funcţia de decrementare ($Y = X-1$) nu apare în definiţia limbajului *LOOP*. Ea poate fi efectuată cu următorul program:

```
Z = 0
Y = 0
LOOP  X
      Y = Z
      Z = Z + 1
END
```

Atenţie! Pentru $X=0$ va rezulta $Y=0$ deoarece operaţiile sînt definite numai pe mulţimea numerelor naturale. □

• **Limbajul maşină al procesorului** sau setul său de instrucţiuni va trebui astfel definit încît să permită cel puţin executarea programelor în limbajul *LOOP*. O *primă variantă* a setului de instrucţiuni va fi următoarea.

**Definiţia 6.5** *Limbajul maşină* (setul de instrucţiuni) al procesorului conţine următoarele instrucţiuni:

- *TAD r,adr*: testează şi decrementează (test and decrement) valoarea din registrul *r* iar dacă *r=0* atunci *PC←adr*;

- *END adr*: marchează sfîrşitul (end) al unei bucle prin *PC←adr*;

- *CLR r*: şterge (clear) conţinutul registrului *r* (*r←zero*);

- *INC r*: incrementează conţinutul registrului *r* (*r←r+1*);

- *MOVE rs, rd*: deplasează (move) conţinutul din registrul sursă (*rs*) în cel destinaţie (*rd*).

Cu *PC* s-a notat un registru privilegiat (program counter) ce permite parcurgerea programului în memoria *RAM* prin adresa pe care o conţine. De regulă valoarea din *PC* se incrementează. Excepţie face acţiunea instrucţiunii *END* şi uneori a instrucţiunii *TAD*. □

Formatul instrucţiunilor este prezentat în figura 6.42. În memoria *RAM*, instrucţiunea *TAD* ocupă două cuvinte de *16* biţi, poziţionate la adrese succesive. Primul reprezintă *codul instrucţiunii* iar cel de al doilea adresa de salt. Codul instrucţiunii este format din trei cîmpuri. Primul codifică tipul instrucţiunii (*TAD*) fiind *000*, iar în următoarele două se află numele binar asociat registrului *r* (apariţia dublată va fi justificată de realizarea concretă a schemei procesorului). Similar apare codul instrucţiunii *END*.

Modul în care este încărcat în *RAM* programul asociat unei bucle *LOOP* este reprezentat în figura 6.43. Bucla ocupă spaţiul cuprins între adresele *#03FA* şi *#0414*, începe cu o instrucţiune *TAD* şi se încheie cu una *END*. Dacă în *TAD* testul dă *r≠0*, atunci se sare locaţia următoare şi se execută instrucţiunea de la adresa *#03FC*, altfel se sare la adresa *#0415*, ieşindu-se din buclă. Instrucţiunea *END* determină reluarea buclei deci saltul la *#03FA*, adresă scrisă în locaţia imediat următoare celei în care se află codul *END* (*#0414*).



Instrucţiunea *CLR r* este codificată prin *010* pe poziţiile cele mai semnificative ale codului instrucţiunii. Urmează două cîmpuri de patru biţi pentru specificarea registrului *r* ce se şterge. Similar este tratată instrucţiunea *INC r*. Pentru instrucţiunea *MOVE rs, rd* registrele sursă şi destinaţie sînt specificate prin două cîmpuri distincte de cîte patru biţi.

Orice program scris în limbajul *LOOP* poate fi transcris în limbajul maşină anterior specificat.

**Exemplul 6.23** Programul scris în *Exemplul 6.21* se transcrie în limbaj maşină asociind numelui *X* conţinutul registrului *R2*, lui *Y* pe *R1*, iar lui *Z* pe *R0*. Rezultă:

```
        MOVE    R1, R0
LOOP    TAD     R2, GATA
        INC     R0
        END     LOOP
GATA    MOVE    R0, R0,
```

folosind, în această etapă, pentru adrese etichete (*LOOP, GATA*).

Dacă presupunem acest program implantat în memoria *RAM* începînd cu adresa *00FF*, el va avea următoarea reprezentare binară (în prima coloană se află adresa în hexa iar în cea de a doua datele din *RAM*):

```
00FF  100.0001.0000.xxxx
0100  000.0010.0010.xxxx
0101  0000.0001.0000.0101
```

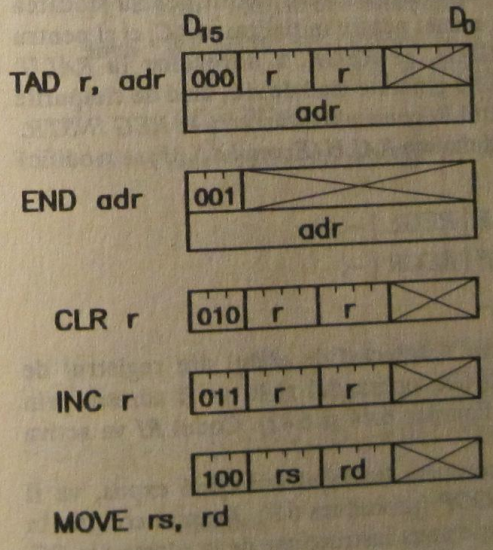



Figura 6.42

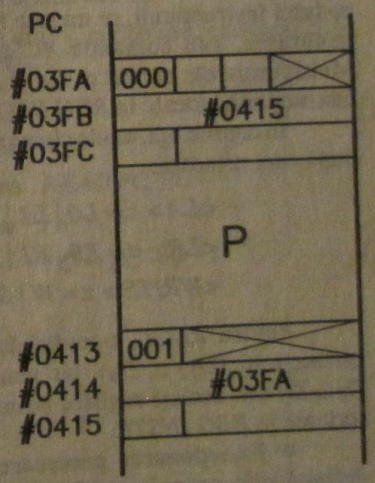



Figura 6.43

*0102* *011.0000.0000.xxxx*

*0103* *001.xxxxxxxxxxxx*

*0104* *0000.0001.0000.0000*

*0105* *100.0000.0000.xxxx.* □

Am pus în evidenţă trei niveluri de specificare a funcţionării unui procesor:

- nivelul limbajului *LOOP*,
- nivelul limbajului maşină,
- nivelul codului binar din *RAM*.

Nu putem să nu remarcăm că pe măsură ce ne apropiem de structura fizică, scrierea programului se face într-o formă din ce în ce mai "neprietenoasă".

• **Structura procesorului** va suferi unele modificări impuse de o modalitate eficientă de interpretare a instrucţiunilor. Pornim de la întrebarea: cîte instrucţiuni distincte conţine limbajul maşină anterior definit? Deoarece rezervăm un registru intern pentru *PC* (să spunem registrul *RF*) rămîn disponibile pentru lucru *15* registre. Vom defini, în consecinţă, o instrucţiune *END*, cîte *15* instrucţiuni de tip *TAD, CLR* şi *INC* şi *15×15=225* instrucţiuni de tip *MOVE*. Rezultă *271* de instrucţiuni distincte la nivelul procesorului. Ar fi absurd să asociem cîte o secvenţă de comenzi fiecărei instrucţiuni. Este lipsit de sens ca pentru *CLR R1* şi *CLR R8* să scriem secvenţe distincte de comenzi. Convenim să trimitem către *AC* numai primii trei biţi ai instrucţiunii iar pe ceilalţi utili să-i transferăm direct către *RALU* prin intermediul unor *MUX*-uri. Va rezulta schema din figura 6.44, unde au fost reprezentate numai completările ce se aduc unei scheme de procesor pentru a suporta noile solicitări. Astfel, *AC* este un automat de control iniţial, de tipul celui din figura 6.36, registrul suplimentar, *REG INSTR*, este folosit pentru stocarea codului instrucţiunii, ce nu este folosit numai pentru iniţializarea *AC*, ci şi pentru generarea, sub comanda *AC* prin *REGL* şi *REGR*, a adreselor la *RALU*. Multiplexoarele permit ca *LA* şi *RA* să fie generate cînd de *AC*, cînd de cîmpurile instrucţiunii stocate în *REG INSTR*. Bitul *RI* comandă încărcarea în *REG INSTR*.

În consecinţă, unele cîmpuri din definirea *RALU* (*Exemplul 5.33*) se modifică după cum urmează:

$$\langle LA \rangle ::= L0 \mid L1 \mid \ldots \mid LF \mid REGL \mid -,$$
$$\langle LR \rangle ::= R0 \mid R1 \mid \ldots \mid RF \mid REGR \mid -,$$
$$\langle WRITE \rangle ::= W \mid RI \mid - .$$

Pentru *REGL*, operandul left va fi selectat de codul din registrul de instrucţiuni ($D_{12}$ ... $D_9$), iar pentru *REGR*, operandul right va fi adresat prin $D_8$... $D_5$ din acelaşi registru (conform figurilor 6.44 şi 6.42). Codul *RI* va activa scrierea în *REG INSTR*.

• **Funcţionarea procesorului**, conform limbajului maşină expus, va fi definită prin procedura *PROCESOR LOOP* (procedura 6.3). Aceasta conţine în principal o buclă infinită pe care se citeşte noua instrucţiune de la adresa din *PC*, se decodifică şi se execută, urmînd citirea unei noi instrucţiuni.



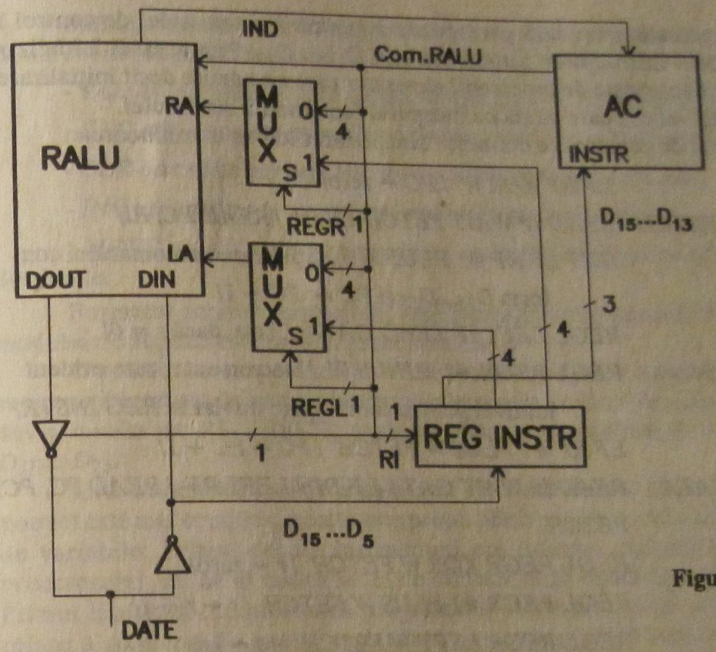



Figura 6.44

**Procedure** *PROCESOR LOOP*

*PC ← zero* / Registrul *F* este şters, el va funcţiona ca *PC*/

**loop** *READ PC, RI* / Citeşte de la adresa din *PC* şi depune rezultatul citirii în *RI (REG INSTR)*/

*PC ← PC + 1*

**case** *: $D_{15}$ ... $D_{13}$ = 000 :* **if** *r = zero*

**then** *READ PC, PC*

**else** *r ← r - 1*

*PC ← PC + 1*

**endif**

*: $D_{15}$ ... $D_{13}$ = 001 : READ PC, PC*

*: $D_{15}$ ... $D_{13}$ = 010 : r ← zero*

*: $D_{15}$ ... $D_{13}$ = 011 : r ← r + 1*

**else** *: rd ← rs*

**endcase**

**repeat**

**end** *PROCESOR LOOP*

**Procedura 6.3**

*Case*-ul din această procedură corespunde *inițializării* automatului de control în funcție de codul instrucțiunii format din biții $D_{15}$ ... $D_{13}$. Prin această inițializare *procesorul* se deosebește de *procesorul elementar* care nu admite decît inițializarea dată de *RESET*-ul ce apare odată cu începutul funcționării sistemului.

Secvența de comenzi ce definește funcționarea *AC* este următoarea:

```
000          LF RF XOR W  / PC ← zero/
001   FETCH  READ IF WAIT FETCH LF RI  / READ PC, RI/
002          INIT LF RF #1 PLUS W  / Inițializarea automatului con-
                 form D15...D13 și PC ← PC + 1/
003          REGL LEFT IF ZERO GATA  / Test, dacă r = 0/
004          REGL REGR #1 MINUS W  / Decrement r; este evident
                 motivul pentru care r apare dublat în REG INSTR/
005          LF RF #1 PLUS W FETCH  / PC ← PC + 1/
006   GATA   READ IF WAIT GATA LF DIN LEFT RF  / READ PC, PC/
007          FETCH
008          REGL REGR XOR W FETCH  / r ← zero/
009          REGL REGR #1 PLUS W FETCH  / r ← r + 1/
00A          REGL REGR LEFT W FETCH  / rd ← rs/.
```

Comanda *INIT* va converti codul $D_{15}$ ... $D_{13}$ în adresa în care se află în *ROM*-ul *AC*-ului începutul secvenței asociate instrucțiunii. Prin intermediul unui transcodor vor rezulta pe intrarea *INSTR* (figura 6.36) adresele *ROM*-ului, conform tabelului 6.6.

| Instr. | $D_{15}$...$D_{13}$ | Adr. ROM |
|---|---|---|
| TAD | 000 | #003 |
| END | 001 | #006 |
| CLR | 010 | #008 |
| INC | 011 | #009 |
| MOVE | 100 | #00A |

**Tabelul 6.6**

Memoria *RAM* este folosită numai pentru stocarea programelor. Pentru date avem la dispoziție *15* din cele *16* registre ale *RALU*. Această situație implică o limitare drastică a complexității programelor ce pot fi executate.

• **Un nou set de instrucțiuni** asociat procesorului ar trebui să permită folosirea unui număr "oricît de mare" de nume. Prin "oricît de mare" ar trebui înțeles că numărul de nume să fie comparabil cu capacitatea de stocare a memoriei *RAM*, exterioară procesorului. Numele (variabilele) nu trebuie stocate numai în registrele procesorului, ele ar trebui să se poată stoca și în *RAM*. Se impune deci un nou limbaj mașină.



**Definiția 6.6** Setul de instrucțiuni care asigură rularea unor programe cu un număr "oricît de mare" de variabile este următorul:

- *TAD av, aj*: unde *av* este adresa variabilei iar *aj* adresa de salt;
- *END adr*: unde *adr* este adresa de salt (identic cu setul anterior);
- *CLR adr*: unde *adr* este adresa variabilei ce ia valoarea zero;
- *INC adr*: unde *adr* este adresa variabilei incrementate;
- *MOVE as, ad*: unde *as* este adresa variabilei sursă iar *ad* adresa variabilei destinație.

Formatul acestor instrucțiuni este reprezentat în figura 6.45 iar detalierea modului de acțiune este dată în procedura 6.4. □

O primă observație ce se poate face este aceea că în structura procesorului ce execută acest set de instrucțiuni multiplexoarele pe intrările *LA* și *RA* ale *RALU* devin inutile iar *REG INSTR* poate fi redus la un registru de trei biți pentru $D_{15}$ ... $D_{13}$.

Este evident că modul în care este efectuată fiecare instrucțiune a acestui nou set este mai complex. Acesta este prețul plătit pentru un număr "oricît de mare" de variabile. Primul set de instrucțiuni era orientat exclusiv către registrele procesorului, cel de al doilea la fel de exclusiv către variabile localizate în *RAM*. Primul limitează complexitatea programelor rulate iar cel de al doilea viteza de rulare a acestora. Se poate, deci, căuta un compromis! Acesta va putea fi găsit într-un set de instrucțiuni cu variabile ce pot fi localizate în egală măsură în registrele interne ale procesorului cît și în *RAM*-ul exterior acestuia.

•**Setul de instrucțiuni orientat pe registre și memorie** va rezulta printr-o combinație a celor două anterioare.

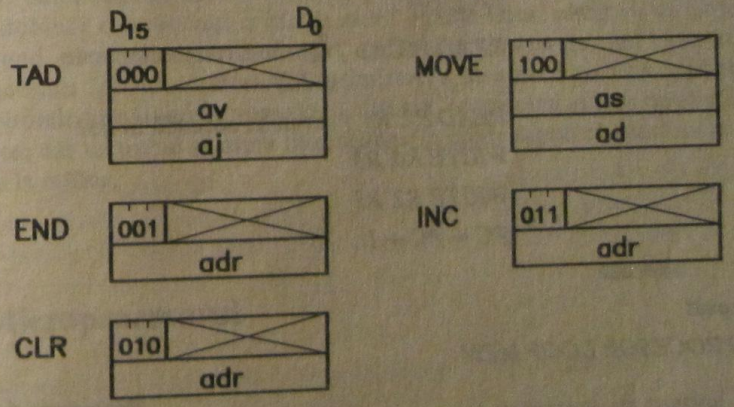



**Figura 6.45**

```
Procedure PROCESOR LOOP MEM
  PC ← zero
loop  READ PC, RI
      PC ← PC + 1
      READ R0, R1  / adresa primei variabile în R0/
      READ R0, R1  / prima variabilă în R1/
      case  : D15 ... D13 = 000 :  if R1 = zero
                                   then   PC ← PC + 1
                                          READ PC, PC
                                   else   R1 ← R1 - 1
                                          WRITE PC, R1  / Scrie la adresa din PC valoarea din R1/
                                          PC ← PC + 2
                                   endif
            : D15 ... D13 = 001 : PC ← R1
            : D15 ... D13 = 010 : R1 ← zero
                                  WRITE R0,R1
                                  PC ← PC + 1
            : D15 ... D13 = 011 : R1 ← R1 + 1
                                  WRITE R0, R1
                                  PC ← PC + 1
            else      PC ← PC + 1
                      READ PC, R2  / adresa variabilei secunde
                                     în R2/
                      READ R2, R3  / variabila secundă în R3/
                      WRITE R0, R3
                      WRITE R2, R1
                      PC ← PC + 1
      endcase
  repeat
end PROCESOR LOOP MEM
```

**Procedura 6.4**



**Definiţia 6.7** Folosind notaţiile celor două definiţii anterioare putem lista următorul set de instrucţiuni :

| | | |
|---|---|---|
| - | *TAD* | *r, adr* |
| - | *TAD* | *av, aj* |
| - | *END* | *adr* |
| - | *CLR* | *r* |
| - | *CLR* | *adr* |
| - | *INC* | *r* |
| - | *INC* | *adr* |
| - | *MOVE* | *rs, r d* |
| - | *MOVE* | *rs, ad* |
| - | *MOVE* | *as, r d* |
| - | *MOVE* | *as, ad* . □ |

Flexibilitatea acestui ultim set de instrucţiuni permite o folosire mai eficientă a resurselor sistemului format din procesor şi memorie. Viteza cu care se poate opera asupra registrelor interne procesorului va putea fi combinată cu capacitatea mare de stocare pentru variabile, pe care o oferă memoria. Intervine în acest moment un element nou: dibăcia cu care se face translatarea programelor scrise în limbajul *LOOP* în programe scrise în limbajul maşină oferit de definiţia 6.7. Dar, aceasta este o problemă ce depăşeşte scopul demersului nostru.

Structura rudimentară de procesor expusă în acest subcapitol este fundamentală. Din punct de vedere tehnic au apărut două tipuri distincte de procesoare: *microprocesorul* şi *procesorul microprogramat*. Microprocesorul oferă *un set fix de instrucţiuni relativ simple* executate cît mai eficient. Procesorul microprogramat permite definirea mai *multor seturi de instrucţiuni* în care fiecare instrucţiune să poată dobîndi *complexităţi oricît de mari*. Aceste două categorii de procesoare corespund unor concepte limită. În realitate procesoarele tind către structuri intermediare într-un demers de optimizare a structurii şi performanţelor. În subcapitolele următoare le vom studia distinct, accentuînd asupra caracteristicilor specifice; dar va trebui să avem întotdeauna în minte faptul că adevărul se află pe undeva la mijloc.

## 6.6 Microprocesorul

În acest moment al expunerii atingem un punct culminant ce conţine în el şi începutul unei decăderi. Ca realizare tehnologică microprocesorul reprezintă o superbă performanţă care, din păcate, sprijină modalităţi de gîndire rudimentare.

Mai mult, apariţia acestui dispozitiv a permis fixarea pentru încă o lungă perioadă de timp a unui stil ce se întrevede, în perspectivă, ca perimat. Microprocesorul încununează tehnologic, începînd din anii '70, concepte ce s-au impus în anii '40. În ultimele două decenii gîndirea funcţională a fost restricţionată de implementări concrete ce presupuneau această fascinantă componentă. Cu toate că microprocesorul a evoluat foarte mult ca arhitectură şi performanţe, nu a putut fi mascată limitarea fundamentală pe care acesta o reprezintă pentru realizarea performantă a unor funcţii complexe din ce în ce mai puternic impuse de mediul tehnologic, social, natural sau uman.

Cu toate aceste critici, nu se poate spune că acest spectaculos dispozitiv nu şi-a găsit o nişă în imensul spectru funcţional pe care electronica digitală încearcă să-l acopere. Există aplicaţii sau subsisteme ce-şi găsesc o concretizare optimă prin utilizarea microcalculatoarelor centrate pe microprocesoare. O abordare lucidă delimitează consistent un spaţiu în care acest dispozitiv se poate impune. Soluţii mai performante, ce presupun instrumente mai puternice, nu fac decît să fixeze mai riguros domeniul de aplicare al microprocesoarelor care nu vor "resimţi" ca pe o decădere tentativa de a nu mai fi considerate instrumente, maşini universale.

### 6.6.1 Arhitectura microprocesorului

Odată cu creşterea în complexitate a sistemelor digitale (dar nu numai a acestora) s-a impus, în primul rînd pentru ţinerea sub control a ansamblului, conceptul de *arhitectură*. Pînă în acest moment am privit sistemele digitale din perspectivă strict *structurală*; am descris, am analizat sau am proiectat configuraţia internă a unui sistem. Optimizarea sistemului am gîndit-o întotdeauna din punctul de vedere al structurii. Odată cu trecerea la sisteme mai complexe începem să resimţim faptul că optimizarea fiecărui subsistem, în raport cu sine numai, nu asigură şi o optimizare a ansamblului. Avem nevoie de un nou concept care să ne permită optimizarea unui sistem complex din perspectiva relaţiilor ce se stabilesc între subsistemele sale. În acest context un subsistem nu intră în discuţie din perspectiva *structurii* sale, ci din perspectiva *funcţiilor* pe care le execută. Ponderea gîndirii funcţionale creşte în defavoarea celei structurale. Astfel se va impune conceptul de *arhitectură*.

**Definiţia 6.8** Fiind date subsistemele $S_1$ şi $S_2$, interconectate pe căi ce transferă configuraţiile binare $X$, de la $S_2$ la $S_1$, şi $Y$, de la $S_1$ la $S_2$, conform reprezentării din figura 6.46, se defineşte arhitectura $A_{12}$ oferită de $S_1$ lui $S_2$ ca fiind

$$A_{12} = \{ F_1, ..., F_n \}$$
$$F_i : X \to Y, \quad i = 1, 2, ..., n,$$

(mulţimea funcţiilor oferite de $S_1$ lui $S_2$). Simetric se defineşte arhitectura $A_{21}$ prin

$$A_{21} = \{ G_1, ..., G_m \}$$
$$G_i : Y \to X, \quad i = 1, 2, ..., m. \square$$



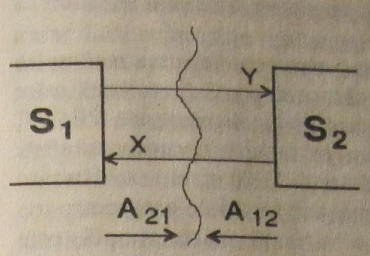



Figura 6.46

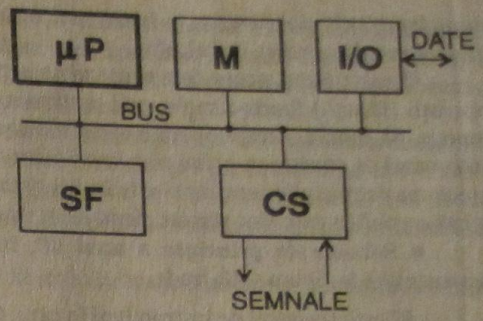



Figura 6.47

Aceeaşi arhitectură poate avea asociate mai multe realizări structurale ce pot diferi prin tehnologia de realizare, prin performanţe sau prin mecanismele de structurare. Avantajul esenţial al gîndirii arhitecturale constă în flexibilitatea realizării şi promovării unor produse tehnice. În primul rînd putem evidenţia faptul că progresele tehnologice pot fi asimilate uneori numai parţial într-un sistem complex, prin reproiectarea unor subsisteme ce se integrează în vechiul sistem fără dificultăţi. Un subsistem ce-şi păstrează nemodificat comportamentul la borne îşi poate modifica structura internă fără a afecta funcţia ansamblului din care face parte. O conectare flexibilă într-un sistem mai complex permite ca, chiar parametrii subsistemului să poată fi îmbunătăţiţi fără a implica modificări în ansamblu. Spre exemplu, un *PE* poate fi cuplat oricînd cu un *FIFO* mai rapid fără să fie necesare modificări structurale sau în secvenţele de comenzi.

Vom defini microprocesorul, pentru început, la nivel arhitectural, urmînd să-i asociem ulterior o variantă structurală.

- **Sistemul în care este interconectat microprocesorul** are, de regulă, configuraţia din figura 6.47. Prin intermediul unui *BUS* de tipul celui din figura 6.34 microprocesorul, *µP*, este conectat la o memorie, *M*, la sistemul de intrare ieşire, *I/O*, la o serie de structuri funcţionale, *SF*, ce realizează performant funcţii critice din sistem şi la un bloc ce asigură conversia de semnale, *CS*. În funcţionarea curentă şi într-o configuraţie minimală sînt suficiente *µP*, *M* şi *I/O*. Selectarea subansamblurilor de pe *BUS* este făcută, de către *µP*, prin codul emis pe calea de adrese. Transferul către şi de la oricare dintre acestea se face asimilîndu-le cu locaţii distincte de memorie. Fiecare dispozitiv conectat pe *BUS* este văzut de *µP* ca o locaţie sau ca mai multe locaţii de memorie distincte.
- **Rolul memoriei conectate pe BUS-ul µP-ului** este mai complex decît s-a întrevăzut în discuţia despre *PE*, unde era folosită numai pentru a stoca datele de prelucrat şi rezultatele. În cazul *µP*-ului memoria *M* stochează şi secvenţe de *instrucţiuni* (coduri de comandă) structurate sub formă de *programe*. Să explicăm!

În cazul unui *PE* schimbarea funcţiei efectuate necesita reproiectarea automatului de control, *AC*, care primea din exterior o singură comandă: *RESET*. Se pune însă problema conceperii unui procesor care să poată fi folosit la realizarea

unor funcţii cît mai diverse, la limită este necesar un sistem care să fie capabil să efectueze orice proces de calcul.

O soluţie este aceea de a construi un procesor care poate efectua mai multe operaţii (funcţii) foarte simple, prin combinarea cărora să poată fi realizată *orice* funcţie. Să numim aceste operaţii simple: *instrucţiuni*. În loc de comanda *RESET*, automatul de control va primi *codul unei instrucţiuni*, iar după ce o va executa se va preocupa de recepţionarea unei alte instrucţiuni. De unde? Din memoria *M* în care există o zonă în care sînt stocate şiruri de instrucţiuni sub formă de programe.

• **Schema de principiu a unui** $\mu$P, într-o variantă foarte simplă, este reprezentată în figura 6.48, unde:

- *RI* este registrul de instrucţiuni în care, de pe calea de date, se poate înscrie codul instrucţiunii curente;

- *RALU* are o configuraţie obişnuită, să spunem cea din figura 5.97, ieşirea *ADR'* provenind de la *LDOUT* al *RAM* din aceeaşi figură;

- *AC* este un automat de control a cărui structură şi organigramă de funcţionare o vom detalia într-un paragraf ulterior.

Un ciclu complet de funcţionare al unui $\mu P$ cuprinde două intervale distincte:

- citirea instrucţiunii din memorie şi stocarea în *RI*,

- executarea instrucţiunii printr-o secvenţă de comenzi realizate sub controlul *AC*, conform instrucţiunii din *RI*.

• **Completitudinea setului de instrucţiuni** asigură posibilitatea realizării oricărui proces de calcul, în sensul că se poate efectua orice funcţie definită pe o structură de date stocabilă în memorie, cu valori tot într-o structură de date localizată în aceeaşi memorie. Seturile curente de instrucţiuni sînt acoperitor definite, în sensul că sînt definite mai multe instrucţiuni decît cele strict necesare completitudinii, pentru a asigura facilităţi de programare prin care *performanţele* $\mu P$ să crească (a se vedea în acest sens setul de instrucţiuni dat de *Definiţia 6.7* în comparaţie cu cel dat de *Definiţia 6.6*). Ce înseamnă ca un $\mu P$ să fie performant? Este o întrebare la care nu se poate oferi un răspuns complet în acest moment. Pe scurt, o *arhitectură performantă* este una care poate fi folosită *flexibil* la rezolvarea *rapidă* a unor probleme *complexe* afectînd, în acest sens, o *capacitate minimă de memorie RAM*.

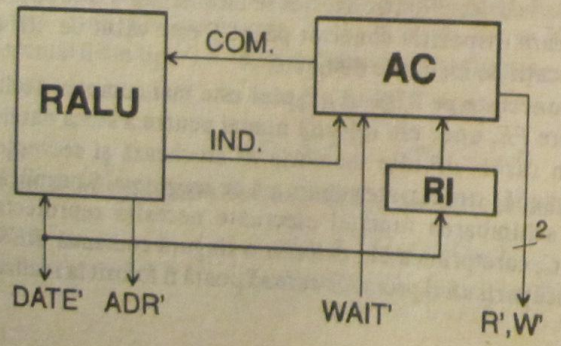



**Figura 6.48**



Arhitectura microprocesoarelor a evoluat mult în ultimele două decenii, chiar dacă principiile fundamentale nu s-au modificat. În acest curs vom prezenta numai *un set de instrucţiuni relativ simplu*, dar complet, *prin care urmărim ilustrarea principiilor de realizare, funcţionare şi utilizare a microprocesoarelor*.

• **Limbajul maşină, sau setul de instrucţiuni al microprocesorului**, va fi definit compact, utilizînd formalismul *BNF*. Structura preliminară pe care o avem în vedere este de tipul celei din figura 6.44; ea va fi concretizată şi eventual completată în paragrafele următoare, dar pentru înţelegerea definiţiei limbajului maşină este suficientă.

**Definiţia 6.9** Instrucţiunile limbajului maşină asociate microprocesorului sînt:

*<Instrucţiune>* ::= *<Funcţie>* | *<Salt>* | *<Transfer>* | *<Control>*

*<Funcţie>* ::= *<Func><Left>*[*<Right>*]*<Result>* / Între paranteze drepte apar cîmpuri opţionale /.

*<Salt>* ::= [ *IF* [*NOT*]*<Test>*]*<Jmp><Adr>*

*<Transfer>* ::= *LOAD <Adr>,<Dest>* | *STORE <Sursă>,<Adr>* | | *MOVE <Sursă>,<Dest>* | *CONST <Val>,<Dest>*

*<Control>* ::= *HALT* | *NOP* | *EI* | *DI*

*<Func>* ::= *AND* | *OR* | *XOR* | *PLUS* | *MINUS* | *INC* | *DEC* | *SHR*

*<Left>*::=*<Right>*::=*<Result>*::=*<Adr>*::=*<Sursă>*::=*<Dest>*::= = *RO* | *R1* | *R2* | *R3* | *R4* | *R5* | *R6* | *R7* | *R8* | *R9* | *R10* | *R11* | *R12* | *R13* | *SP* | *PC*

*<Test>* ::= *CR* | *SGN* | *OVFL* | *ZERO* | *EQ* | *PAR*

*<Jmp>* ::= *JMP* | *CALL* | *RET*

*<Adr>* ::= *<Val>* ::= *#0* | *#1* | ... | *#FFFF* □

Pentru înţelegerea setului de instrucţiuni anterior expus vom da, în continuare, o serie de exemple.

**Exemplul 6.24** Exemple de instrucţiuni:

- *XOR R5, R2, R8*: se realizează bit cu bit funcţia *XOR* între conţinuturile registrelor *5* şi *2* rezultatul fiind depus în registrul *8*;

- *SHR R5, R1*: conţinutul din *R5* se deplasează cu o poziţie la dreapta (împărţirea întreagă cu 2) şi rezultatul se depune în *R1*, fără ca registrul *R5* să fie afectat;

- *INC R6, R6*: se incrementează conţinutul registrului *R6*;

- *MINUS R8, R3, R12*: se scade din *R8* valoarea din *R3* şi rezultatul se depune în *R12*;

- *JMP #1010*: *PC* ← *#1010*, salt necondiţionat la adresa *#1010* în zona de program;

- *IF CR JMP #2FA1*: dacă *CR=1* atunci *PC* ← *#2FA1*, altfel, *PC* urmăreşte continuarea programului în aceeaşi zonă;

- *CALL #13*: salt la subrutina ce începe la adresa *#13*, salvîndu-se în stivă adresa de reîntoarcere;

- *RET*: întoarcerea din subrutină încărcînd în *PC* valoarea din vîrful stivei;

- *LOAD R2, R11*: în *R11* se încarcă conţinutul locaţiei de memorie adresată prin *R2*;

- *STORE R7, R0*: conţinutul din *R7* se înscrie în memorie la adresa din *R0*;

- *MOVE R2, R10*: conţinutul registrului *2* se transferă în registrul *10*;

- *CONST #1A2C, R9*: în *R9* se încarcă valoarea *#1A2C*;

- *HALT*: opreşte funcţionarea microprocesorului trecînd în starea de aşteptare a întreruperii;

- *NOP*: instrucţiune în care nu se execută nici o operaţie asupra conţinutului registrelor de lucru;

- *EI*: permite (enable) acţiunea întreruperii în ciclurile ce urmează;

- *DI*: inhibă (desable) acţiunea întreruperilor în ciclurile ce-i urmează (pînă la apariţia unui *EI*). □

*Atenţie!* Secvenţa de instrucţiuni ...*DI*, *HALT*, ... trece microprocesorul într-o stare din care nu mai poate fi scos prin procedee blînde. Va fi necesar un reset general pentru a repune sistemul în funcţie, pentru că în *HALT* este aşteptată o întrerupere ce nu mai poate fi recepţionată.

## 6.6.2 Funcţionarea microprocesorului

Din *Definiţia 6.9* rezultă patru categorii de instrucţiuni. Pentru fiecare vom avea o reprezentare distinctă prin cuvintele stocate în memorie.

• **Formatul instrucţiunilor funcţionale** este reprezentat în figura 6.49*a*. Această clasă de instrucţiuni este caracterizată prin $I_{15} = 1$, urmează cîmpul de trei

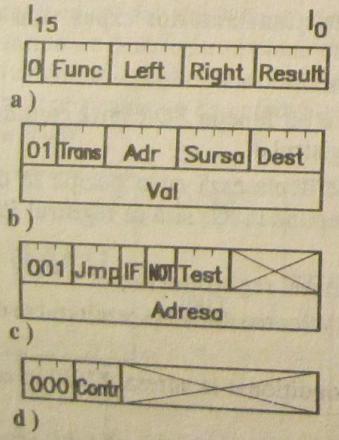



Figura 6.49

| Func | $I_{14}$ | $I_{13}$ | $I_{12}$ |
|---|---|---|---|
| AND | 0 | 0 | 0 |
| OR | 0 | 0 | 1 |
| XOR | 0 | 1 | 0 |
| PLUS | 0 | 1 | 1 |
| MINUS | 1 | 0 | 0 |
| INC | 1 | 0 | 1 |
| DEC | 1 | 1 | 0 |
| SHR | 1 | 1 | 1 |

Tabelul 6.7 a)



biţi ce conţine codul funcţiei conform tabelului 6.7*a*, iar apoi trei cîmpuri de cîte patru biţi prin care se indică cei doi operanzi şi destinaţia rezultatului. Operanzii sînt localizaţi în cele *16* registre ale *RALU* iar destinaţia rezultatului se află în aceleaşi *16* registre.

• **Formatul instrucţiunilor de transfer** este reprezentat în figura 6.49*b* fiind caracterizat prin $I_{15}I_{14} = 01$. Codurile celor patru instrucţiuni de transfer sînt date în tabelul 6.7*b*. Următorul cîmp de patru biţi, $I_{11} \dots I_8$ reprezintă registrul în care se află adresa la care sau de la care se realizează transferul. $I_7 \dots I_4$ reprezintă registrul transferat prin *STORE* sau *MOVE*. $I_3 \dots I_0$ este registrul în care se transferă prin *LOAD* şi *MOVE* sau se încarcă prin *CONST* valoarea din locaţia de memorie imediat următoare (de la adresa *PC+1*).

| Trans | $I_{13}$ | $I_{12}$ |
|---|---|---|
| STORE | 0 | 0 |
| LOAD | 0 | 1 |
| CONST | 1 | 0 |
| MOVE | 1 | 1 |

Tabelul 6.7 b)

| Jmp | $I_{12}$ | $I_{11}$ |
|---|---|---|
| RET | 0 | – |
| CALL | 1 | 0 |
| JMP | 1 | 1 |

Tabelul 6.7 c)

• **Formatul instrucţiunilor de salt** este reprezentat în figura 6.49*c*, fiind caracterizat prin $I_{15}I_{14}I_{13} = 001$. Codurile asociate instrucţiunilor de salt se află în tabelul 6.7*c*. Bitul $I_{10} = 1$ indică faptul că saltul este condiţionat iar $I_9$ specifică dacă condiţia de salt este negată ($I_9 = 1$) sau nu. Indicatorii testaţi sînt codificaţi în tabelul 6.7*d*.

• **Formatul instrucţiunilor de control** este reprezentat în figura 6.49*d* şi este caracterizat prin $I_{15} \dots I_{13} = 000$. Codurile acestor instrucţiuni se găsesc în tabelul 6.7*e*.

| Test | $I_8$ | $I_7$ | $I_6$ |
|---|---|---|---|
| CR | 0 | 0 | 0 |
| SGN | 0 | 0 | 1 |
| OVFL | 0 | 1 | 0 |
| ZERO | 0 | 1 | 1 |
| EQ | 1 | 0 | 0 |
| PAR | 1 | 0 | 1 |

Tabelul 6.7 d)

| Contr | $I_{12}$ | $I_{11}$ |
|---|---|---|
| HALT | 0 | 0 |
| NOP | 0 | 1 |
| EI | 1 | 0 |
| DI | 1 | 1 |

Tabelul 6.7 e)

• **Registrul de stare** va fi folosit pentru stocarea indicatorilor ce determină realizarea salturilor condiţionate. Vom particulariza structura din subcapitolul 4.6 sub forma celei prezentate în figura 6.50 unde transcodorul *TC* realizează o funcţie de tipul celei descrise în tabelul 4.1 şi are condiţionată activarea ieşirilor $O_0, ..., O_5$ de $E = 1$. În această figură este reprezentat numai un singur bistabil de stare. Modul de funcţionare este impus de codul $M_1M_0$ conform tabelului 6.8.

| $M_1$ | $M_0$ | |
|---|---|---|
| 0 | 0 | Noop ( $S_1 S_0 = 00$, bistabilul se încarcă cu propria stare ) |
| 0 | 1 | Actualizare ($S_1S_0 = OX$, în unii bistabili se încarcă indicatorul) |
| 1 | 0 | Încărcare ($S_1S_0 = 10$, se încarcă bistabilii cu Di) |
| 1 | 1 | cod nefolosit |

**Tabelul 6.8**

Conţinutul registrului de stare se actualizează numai atunci cînd este efectuată o instrucţiune de tip funcţional, în ciclul în care funcţia (*Func*), indicată de codul instrucţiunii, se execută. Cu această ocazie numai o parte din biţii de stare se vor modifica. Aceşti biţi vor fi selectaţi de poziţiile egale cu *1* ale configuraţiei binare, $O_5...O_0$, de la ieşirea transcodorului *TC*. De asemenea, uneori conţinutul acestui registru va fi stocat în memoria externă microprocesorului şi va putea fi restaurat prin reîncărcarea din memorie. Acest mecanism este utilizat pentru a permite executarea unor subrutine. Saltul la subrutină presupune salvarea în memorie a conţinutului registrului de stare, iar reîntoarcerea din subrutină cere restaurarea acestuia, prin reîncărcarea cu conţinutul locaţiei de memorie în care a fost salvat. De regulă, conţinutul registrului de stare este salvat într-o stivă (*LIFO*). Deci, $Out_i$ va fi trimis atît către automatul de control, *AC*, cît şi către memoria externă, pe căile bidirecţionale de *Date*. Intrările $D_i$ permit restaurarea, fiind cuplate tot la *Date*.

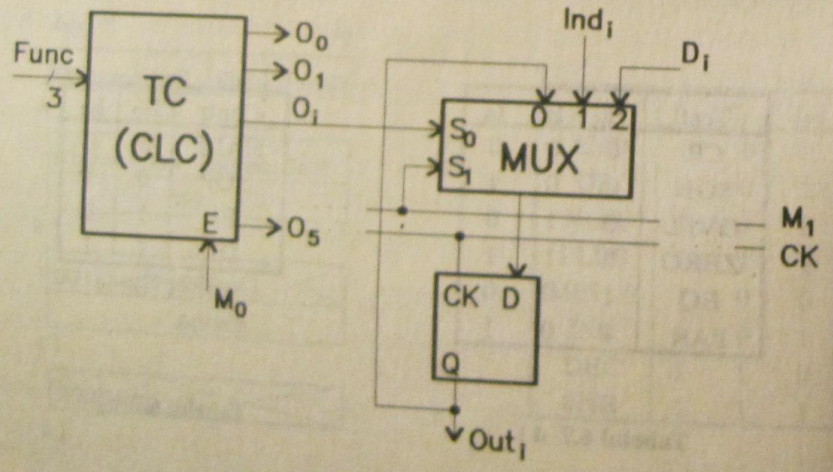



**Figura 6.50**



Codul $M_1 M_0$ este generat de către automatul de control *AC*. Schema este astfel concepută încît pentru $M_1 M_0 = 11$ generează o comportare disfuncţională a registrului de stare. Acest cod este nefolosit în definirea *RS*, dar nu este permisă generarea lui în nici un ciclu de funcţionare a sistemului.

• **Schema de principiu a microprocesorului**, reprezentată în figura 6.51, este formată din următoarele subansambluri:

- *RALU* cu *16* registre de cîte *16* biţi de tipul celui din figura 5.97 sau 5.98 cu modificările şi conexiunile ce se vor impune din descrierea detaliată a funcţionării microprocesorului;

- *AC* (automatul de control);

- *RI* (registrul de instrucţiuni);

- *RS* (registrul de stare);

- *MUX*, un grup de multiplexoare ce asigură o parte din comenzile la bornele *RALU* de la *AC* sau *RI*, sub controlul (selecţia) semnalelor din *AC*.

*Semnalele interne* ce asigură funcţionarea sistemului sînt următoarele:

- *Comenzi*, generate de *AC* către subansambluri sau către exterior;

- *Instrucţiuni*, generate din *RI* către *AC* şi, prin intermediul *MUX*, către *RALU*;

- *Indicatori*, generaţi, de *ALU* din *RALU*, către *RS*;

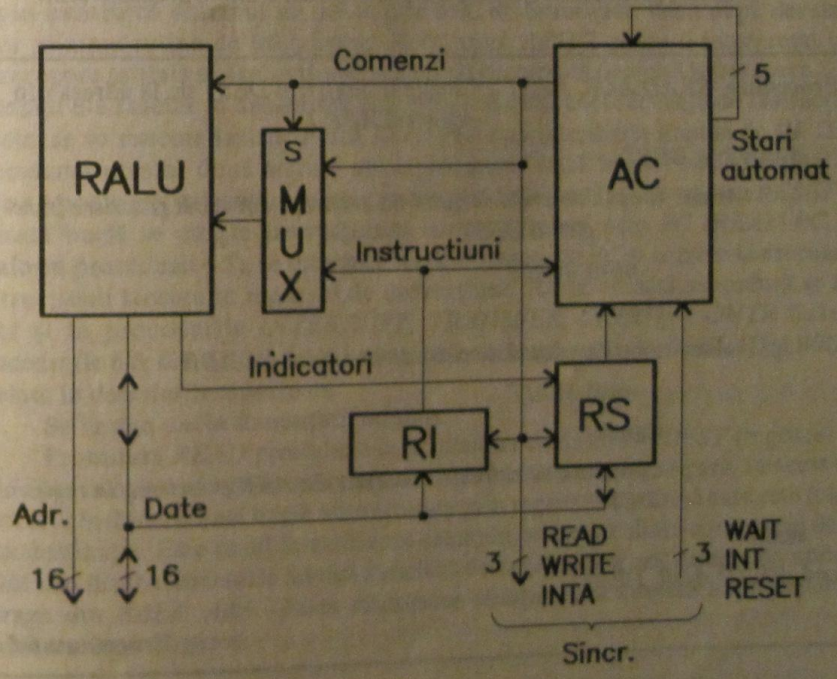



**Figura 6.51**

- *Date* reprezintă un bus intern bidirecţional pe care sînt cuplate *RALU*, *RS* şi *RI* (unidirecţional);

- *Adr*, cale de adrese ce se emite în exterior din *RALU* pentru controlul dispozitivelor de memorare exterioare microprocesorului.

*Semnalele la bornele microprocesorului* sînt următoarele:

- *Adr*, cale unidirecţională;

- *Date*, cale bidirecţională;

- Semnale de sincronizare:

• *READ*, semnal ce comandă citirea de la *Adr* (acţiunea acestui semnal este descrisă prin procedura 6.5);

• *WRITE*, semnal ce comandă scrierea la *Adr* a configuraţiei binare de pe calea *Date* (acţiunea acestui semnal este descrisă prin procedura 6.6);

• *WAIT*, semnal recepţionat ce impune trecerea microprocesorului într-o stare de aşteptare drept urmare a unei comenzi *READ* sau *WRITE* ce nu a fost încă executată;

• *INT*, semnal de întrerupere a cărui recepţionare stopează procesul curent de calcul pentru a da curs unui eveniment exterior sistemului;

• *INTA*, semnal ce indică luarea în consideraţie a semnalului de întrerupere, *INT*; ca urmare a acestui semnal, dispozitivul ce a activat semnalul *INT* va genera pe *Date* o configuraţie binară specifică;

• *RESET*, semnal de iniţializare a funcţionării microprocesorului.

---

**Procedure** *READ ADR, DEST* / Citeşte în registrul *DEST* de la adresa din registrul *ADR*./

**if** *ADR* ≠ *DEST*

**then** **loop** *Comandă de citire de la adresa din ADR şi scriere în registrul DEST*

**until** *WAIT*

**repeat**

**else** **loop** *Comandă de citire de la adresa din ADR*

**until** *WAIT*

**repeat**

*Comandă de citire de la adresa din ADR şi scrierea în registrul DEST*

**endif**

**end** *READ ADR, DEST*

**Procedura 6.5**

---



---

**Procedure** *WRITE ADR, SURSA* / Scrie conţinutul registrului *SURSA* la adresa din registrul *ADR*./

**loop** *Comandă de scriere cu selectarea registrelor pentru ADR şi SURSA*

**until** *WAIT* / Pînă cînd memoria anunţă că scrierea este efectuată/

**repeat**

**end** *WRITE ADR, SURSA*

**Procedura 6.6**

---

• **Funcţionarea microprocesorului** va fi descrisă prin două modalităţi ce se vor completa în a oferi o imagine cît mai limpede. O primă imagine va fi dată printr-un set de *proceduri*, pentru a fixa aspectele funcţionale. Pentru a da şi o imagine asupra înlănţuirii temporale va fi folosită descrierea printr-o *organigramă*.

Procedura *MICROPROCESSOR* (procedura 6.7) descrie funcţionarea globală a microprocesorului odată cu iniţializarea sa prin semnalul *RESET*, care impune resetarea registrului intern *PC* (al *16*-lea registru din *RALU*) şi interzicerea accesului semnalului *INT* în microprocesor. *PC* este resetat pentru că primele instrucţiuni care trebuie executate la activarea sistemului ce conţine microprocesorul se află în memorie începînd de la adresa zero ($0000_H$). *DI* trebuie impus pentru că sistemul nu poate ţine cont de întreruperi decît după derularea unor minime rutine de iniţializare. Altfel spus, *RESET* poate fi interpretat ca o întrerupere iniţială a cărei acţiune nu poate fi perturbată de o altă întrerupere. Cînd sistemul s-a resetat în ansamblul său, prin acţiunea unei secvenţe de instrucţiuni rulate, se va executa instrucţiunea *EI* ce anulează interdicţia impusă de *DI*. După executarea acestor două acţiuni microprocesorul intră într-o "buclă infinită" din care nu poate ieşi sub acţiunea unei instrucţiuni ci numai printr-un nou *RESET*. Pe această buclă se citeşte instrucţiunea curent adresată prin *PC* (*READ PC, RI*, conform procedurii 6.5), se incrementează valoarea din *PC* şi se trece la executarea instrucţiunii stocate în registrul de instrucţiune, *RI*. În această procedură se face apel şi la procedurile *INTERRUPT, TRANSFER, SALT* şi *CONTR* date în procedurile 6.8, 6.9, 6.10 şi 6.11. La acestea se adaugă procedurile *READ* şi *WRITE* apelate în descrierile anterioare.

Se impun unele comentarii minime.

Procedura *READ* presupune cazul distinct cînd *ADR* = *DEST* (registrul din *RALU* ce conţine adresa este acelaşi cu cel în care se citeşte). Apare, în acest caz, pericolul închiderii unei bucle necontrolate prin registrul de adresă care este folosit şi ca destinaţie. Este cazul în care, spre exemplu, se citeşte dintr-o memorie de tip *ROM* sau dintr-o memorie statică suficient de rapide. Oricum, *atenţie*, de unde se extrage din *RALU Adr*! Lăsăm cititorului soluţionarea corectă a problemei şi evaluarea consecinţelor.

```
Procedure MICROPROCESSOR
    PC ← zero  / Se resetează registrul program counter, PC /
    DI  / Este blocat accesul semnalului de întrerupere, INT/
    loop  READ PC, RI  / Citirea instrucţiunii curente în RI de la adresa con-
                         ţinută de registrul PC/
          PC ← PC plus 1  / Incrementarea PC-ului/
          if HALT  / Dacă instrucţiunea adusă în RI este HALT/
              then    loop / Buclă pe care nu se execută nimic în aşteptarea INT/
                      until INT
                      repeat
                      INTERRUPT  / Procedură de tratare a semnalului INT/
              else    case  : I15 = 1 : Execută funcţia din RI (I14I13I12)
                                asupra operanzilor specificaţi prin
                                I11 ... I8 şi I7 ... I4 şi memorează
                                rezultatul în registrul specificat
                                prin I3 ... I0.
                            : I14 = 1 : TRANSFER
                            : I13 = 1 : SALT
                            else CONTR
                      endcase
          endif
          if  INT
              then INTERRUPT
          endif
    repeat
end MICROPROCESSOR
```

**Procedura 6.7**

Procedura *SALT* presupune alocarea unei zone în memoria externă pentru organizarea unei stive avînd pointer-ul de stivă (*SP*), adică adresa vîrfului stivei, stocat în al *15*-lea registru din *RALU*. Valoarea *SP* va fi setată în cadrul programului de iniţializare, de care am amintit, care începe la adresa $0000_H$. Acesta este unul din motivele pentru care apare *DI* drept urmare a semnalului de *RESET*; înainte de recepţionarea unei întreruperi trebuie definită, cel puţin, zona unde se organizează stiva prin încărcarea registrului *SP* cu o valoare bine precizată. La *CALL* (chemarea



```
Procedure INTERRUPT
    SP ← SP minus 1, DI  / Decrementarea pointer-ului de stivă şi DI./
    WRITE SP, PC  / Salvarea, în stivă, a conţinutului registrului PC./
    SP ← SP minus 1
    WRITE SP, RS  / Salvarea registrului de stare./
    INTA, READ bus, PC  / Achitarea întreruperii cu semnalul INTA şi citirea,
                          de pe bus în PC, a codului setat din exterior, drept
                          urmare a semnalului INTA, de către dispozitivul ce
                          a generat INT. Acest cod va fi folosit pentru a adresa rutina de tratare a întreruperii./
end INTERRUPT
```

**Procedura 6.8**

```
Procedure TRANSFER  / Subsetul instrucţiunilor de TRANSFER/
    case  : I13I12 = 00 : WRITE Adr, Sursă  / Scriere la adresa din registrul specificat prin I11 ... I8 a conţinutului
                                              registrului specificat prin I7 ... I4./
          : I13I12 = 01 : READ Adr, Dest  / Citire în registrul specificat prin
                                            I3 ... I0./
          : I13I12 = 10 : READ PC, Dest  / Citire de la adresa din PC./
                          PC ← PC plus 1
          else Dest ← Sursă  / Registrul specificat prin I3 ... I0 ia valoarea registrului specificat prin I7 ... I4./
    endcase
end TRANSFER
```

**Procedura 6.9**

unei subrutine) este obligatorie salvarea adresei de retur în programul principal (*PC+1*) şi salvarea conţinutului registrului de stare. Dacă utilizatorul consideră necesar va putea salva, în primele instrucţiuni din subrutină, şi conţinutul registrelor *0, ..., 13* sau al unora din ele.

Procedura *INTERRUPT* este de fapt un *CALL* la adresa generată pe *Data*, de către dispozitivul ce a generat întreruperea, drept urmare a semnalului *INTA*.

```
Procedure SALT / Subsetul instrucţiunilor de SALT/
  if    I'10 + T = 1  / Dacă saltul este necondiţionat sau condiţionat de o
                        condiţie îndeplinită./
        then if I12 = 1  / Dacă nu este instrucţiunea RET./
                then    if I11 = 0  /Dacă este instrucţiunea CALL (salt la
                                      subrutină./
                        then SP ← SP minus 1  / Decrementarea indicatorului vîrfului stivei./
                             WRITE SP, PC plus 1  / Salvarea adresei de retur./
                             SP ← SP minus 1
                             WRITE SP, RS  / Salvarea registrului de stare./
                        endif
                        READ PC, PC
                else    READ SP, RS  / Restaurarea registrului de stare./
                        SP ← SP plus 1
                        READ SP, PC / Restaurarea program counter-ului./
                        SP ← SP plus 1
             endif
        else if I12 = 1
                then PC ← PC plus 1  / Este depăşită adresa de salt./
             endif
  endif
end SALT
```

**Procedura 6.10**

```
Procedure CONTR / Subsetul instrucţiunilor de CONTROL, fără HALT./
  if    I12 = 1  / Dacă nu este instrucţiunea NOP, pentru că HALT nu poate fi./
        then if I11 = 0  / Dacă este EI./
                then Setarea bistabilului ce deschide poarta de acces a semnalului INT
                else Resetarea bistabilului ce deschide poarta de acces a semnalului INT.
             endif
  endif
end CONTR
```

**Procedura 6.11**



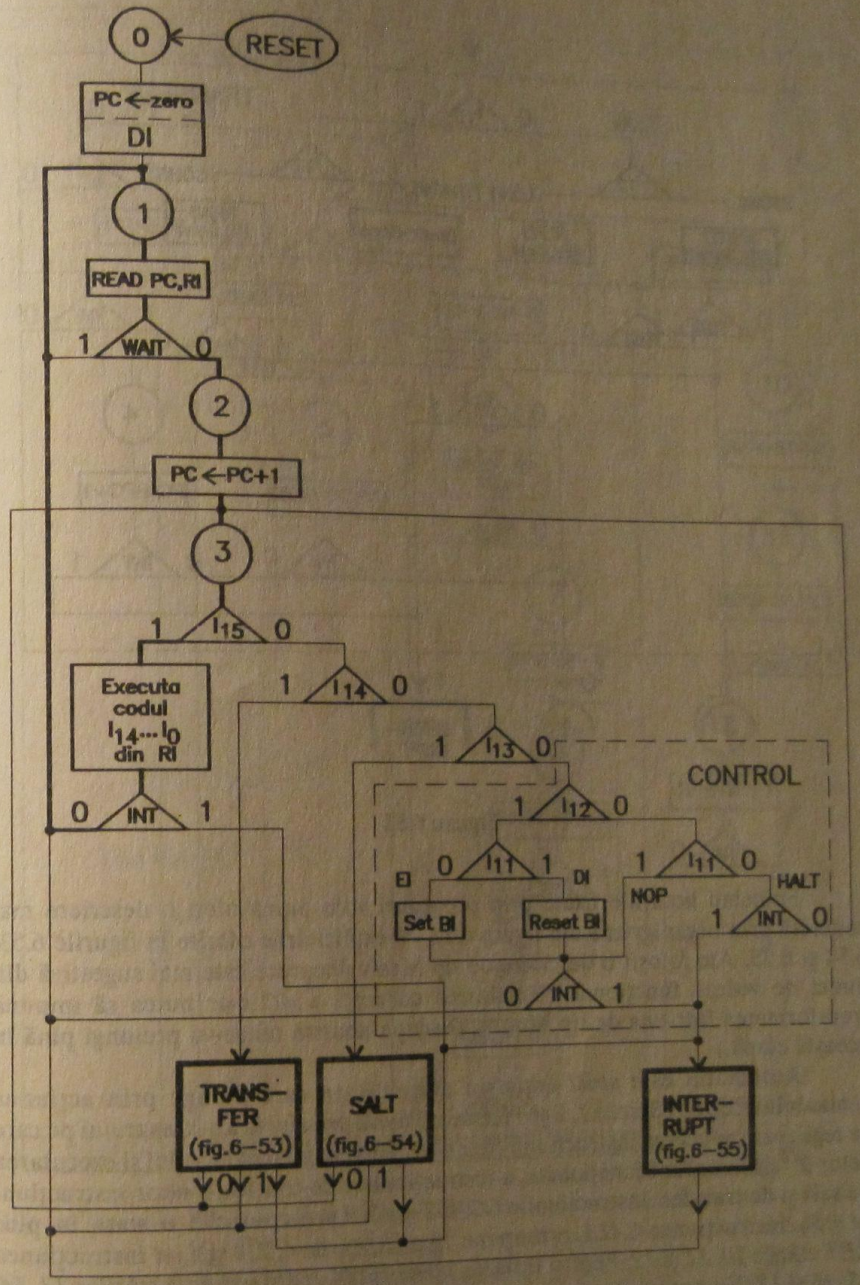



**Figura 6.52**

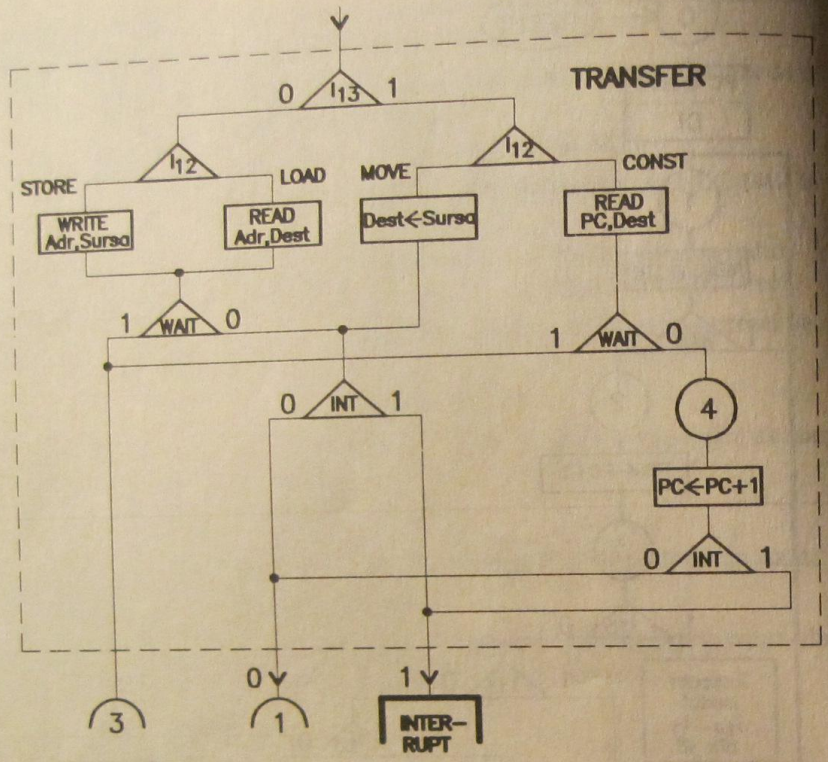



**Figura 6.53**

Folosind notaţiile fixate prin proceduri vom putea oferi o descriere mai concretă prin organigrama din figura 6.52, cu explicitările oferite în figurile 6.53, 6.54 şi 6.55. Am folosit o descriere de tip Mealy deoarece este mai sugestivă din punct de vedere funcţional. (Realizarea concretă a *AC* s-ar putea să impună transformarea într-una de tip Moore. Discuţia noastră nu se va prelungi pînă în această etapă.)

Automatul este *strict iniţial* în starea *0*, în care ajunge prin acţiunea semnalului *RESET*. Stările *1, 2* şi *3* formează *bucla principală* a automatului pe care se realizează aducerea instrucţiunii (fetch-ul), incrementarea *PC*-ului şi executarea celor $2^{15}$ instrucţiuni funcţionale, a instrucţiunilor de control, a unor instrucţiuni de salt şi de transfer. Instrucţiunile *CONST* şi *JMP* presupun cîte o stare în plus (*4* şi *5*). Instrucţiunea *CALL* presupune, în plus, stările *6, 7, 8* şi *9* iar instrucţiunea *RET* stările *10, 11* şi *12*. Pentru tratarea întreruperii apare secvenţa stărilor *13, 14, 15, 16* şi *17*. Nu putem să nu remarcăm această disproporţionată distribuţie; *cîteva instrucţiuni sînt realizate utilizînd majoritatea stărilor automatului de control.*



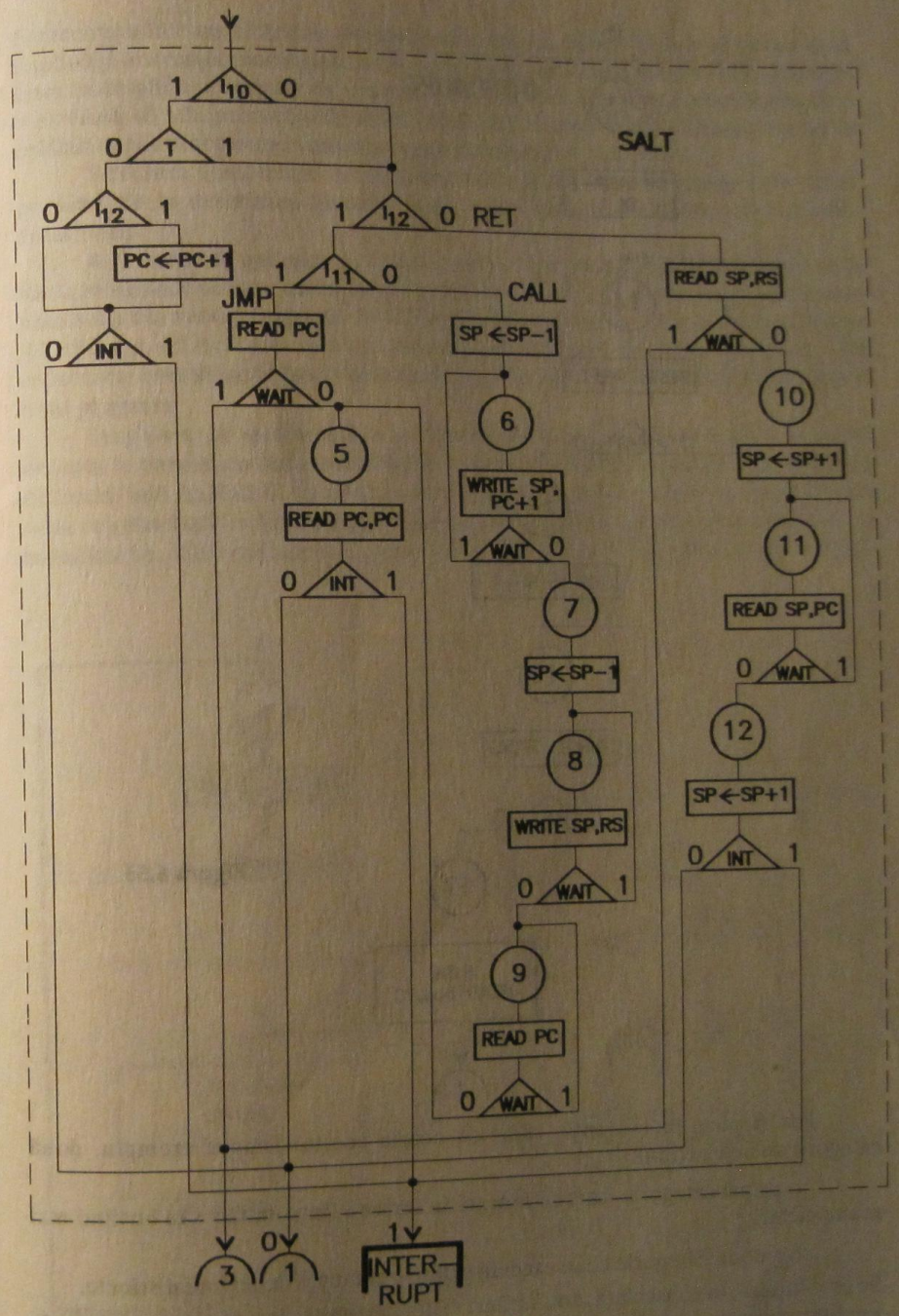



**Figura 6.54**

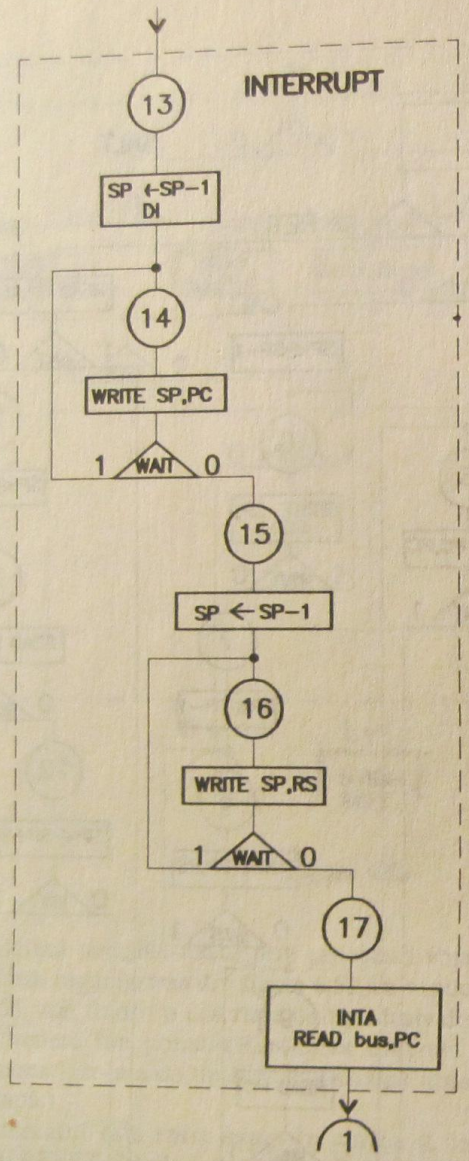



**Figura 6.55**

Pot fi puse în evidenţă, chiar cu ocazia acestui simplu exemplu, două categorii de instrucţiuni:

- o primă categorie, care se execută în starea *3*, într-un tact sau buclînd mai multe cicluri;

- o a doua categorie ce se execută printr-o secvenţă de acţiuni distincte.

Se evidenţiază, în consecinţă, două stiluri distincte, limită, de a concepe arhitectura unui microprocesor. Primul presupune un set de instrucţiuni relativ simple ce pot



fi executate într-un ciclu sau într-un număr mic de cicluri. Cel de al doilea mod implică şi instrucţiuni complexe ce se execută într-un număr mare de stări. Nu vom intra în detaliile presupuse de această distincţie; vom menţiona numai faptul că la o extremă se află procesoarele *RISC* (*R*educed *I*nstruction *S*et *C*omputer) iar la celălalt capăt procesoarele microprogramate orientate funcţional.

Structura completă a microprocesorului poate fi acum desenată de cititor pornind de la descrierea anterioară şi de la structurile definite în secţiunile anterioare.

● **O variantă optimizată** a microprocesorului poate fi gîndită pornind de la *bucla principală* pe care execuţia instrucţiunilor o solicită la maximum. Reducerea unei stări din suborganigrama *INTERRUPT*, spre exemplu, ar minimiza structura *AC* dar nu ar modifica practic deloc performanţele microprocesorului. Dar reducerea unei stări pe bucla principală poate creşte sensibil viteza de execuţie a unui program.

Acţiunea de incrementare a *PC*-ului din starea *2* (figura 6.52) s-ar putea executa în paralel cu cea comandată în starea *3*, dacă am dispune de *PC* ca resursă distinctă faţă de *RALU*. În acest caz bucla principală s-ar reduce la una de forma celei reprezentate în figura 6.56. Pentru aceasta structura microprocesorului se va modifica în conformitate cu figura 6.57, unde au fost adăugate un numărător

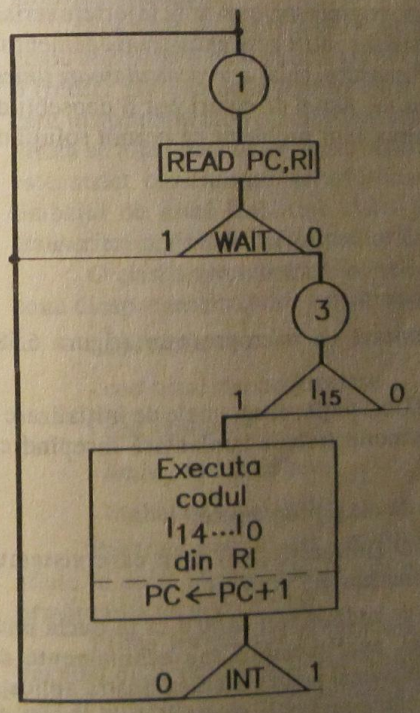



**Figura 6.56**

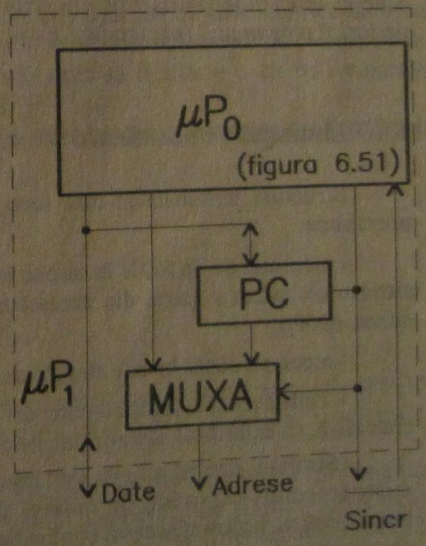



**Figura 6.57**

presetabil, notat *PC*, şi un multiplexor (*MUXA*) ce poate genera *Adr* în exterior, din *PC* sau dintr-un registru din *RALU*, sub comanda automatului de control *AC*. Varianta $\mu P_1$ va funcţiona cu bucla principală realizată pe două stări faţă de $\mu P_0$ care bucleză, pentru aceeaşi funcţionalitate, pe trei stări. Ne aflăm în faţa unui foarte justificat proces de segregare funcţională, registrul *PC* din *RALU*, în $\mu P_0$, devenind numărătorul presetabil *PC* în $\mu P_1$.

Ne putem pune întrebarea dacă nu cumva şi *SP* ar putea să se separe din *RALU* sub forma unui numărător up-down presetabil. Am avea de a face cu un proces nejustificat de segregare funcţională, deoarece nu se va putea constata o creştere semnificativă a vitezei de rulare a programelor pe un microprocesor astfel modificat. Raritatea execuţiilor instrucţiunilor *CALL* şi *RET*, precum şi raritatea apariţiei semnalului *INT*, fac astfel încît manipularea mai eficientă a conţinutului registrului *SP* să nu fie semnificativă în calculul timpului de execuţie al unei aplicaţii.

- **Microprocesoarele concret realizate** posedă arhitecturi (seturi de instrucţiuni) şi structuri interne mai complexe, dar în principiu sînt gîndite în maniere ce au fost exemplificate de microprocesorul simplu anterior descris. Produsele astăzi curente pe piaţă sînt gîndite pe *32* de biţi (spaţiu de adresare de *4* gigacuvinte), au facilităţi aritmetice mult mai complexe (chiar operaţii de înmulţire şi împărţire în virgulă mobilă) şi mecanisme de adresare mai sofisticate. De asemenea, tehnologiile moderne permit integrarea pe acelaşi cip şi a unor funcţii ce de regulă sînt gîndite în exteriorul unui microprocesor, cum ar fi: interfeţe seriale sau paralele, memorii *ROM* sau *RAM*, unităţi pentru gestiunea (managementul) memoriei, memorii cache ş.a. Se pot astfel constitui chiar *microcalculatoare unicip* cu memorie relativ mică şi periferie minimală. Astfel de cipuri pot fi deosebit de utile în aplicaţii industriale pentru rezolvarea unor probleme ce permit soluţii de tip *logică programată* (paragraful 6.6.4).

### 6.6.3 Limbajul de asamblare

Structura minimală în care funcţionează un microprocesor (figura 6.58) presupune:

- memorie de tip *ROM* în care se află, cel puţin, programele de iniţializare a sistemului; măcar o parte din această memorie trebuie implantată începînd cu adresa $0000_H$;

- memoria *RAM* în care sînt stocate datele şi programele;

- rudimentare echipamente de *I/O* (intrare-ieşire) prin care sistemul comunică, cu exteriorul, sincronizat de semnalele *INT* şi *INTA*.

(Sistemul descris nu mai este unul de ordinul trei pentru că în bucla unui microprocesor, ca sistem de ordinul trei, apar memorii sau echipamente de intrare-ieşire. Folosim această configuraţie numai pentru a putea ilustra aplicaţii elementare ale microprocesorului, fără a emite pretenţia studierii unui sistem de ordin superior lui trei.)



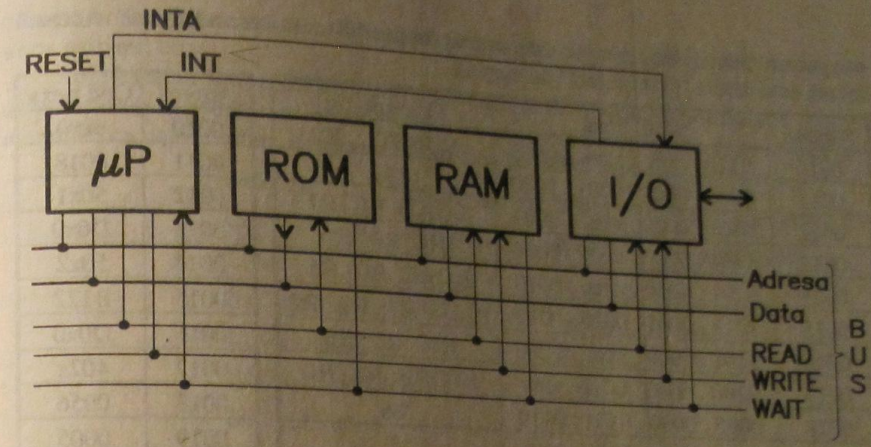



**Figura 6.58**

Pentru ca un astfel de sistem să funcţioneze este absolut necesară scrierea unor programe care să poată fi executate din *ROM* sau *RAM*. În consecinţă sînt necesare procedee prin care aceste programe să fie scrise cît mai eficient şi cît mai controlabil. Evoluţia limbajelor de programare este impulsionată de aceste motivaţii esenţiale.

Formele de reprezentare ale programelor în interiorul structurii nu sînt deloc adecvate şi manipulării lor de către utilizatorul uman. Din acest motiv există o trecere gradată de la *formele interne de reprezentare* ale unui program la cele în care utilizatorul îşi reprezintă *algoritmul* pentru soluţionarea unei probleme. La o limită se află *codul binar*, iar la cealaltă limită *limbajele de nivel înalt*. Codul binar este strict condiţionat de arhitectura şi structura sistemului ce-l execută, iar limbajul de nivel înalt este astfel gîndit încît să faciliteze o cît mai expresivă transcriere a gîndului programatorului.

O clasificare sumară a formelor de reprezentare a programelor (detaliată în zona dinspre maşină) este următoarea:

- cod binar

- cod octal sau hexazecimal

- cod simbolic

- *limbaj de asamblare*

- limbaj de nivel înalt.

Pentru exemplificare vom considera programul ce adună numerele (*86* şi *5*) aflate în două locaţii succesive de memorie (*#18* şi *#19*) şi depune rezultatul în următoarea locaţie (*#1A*).

Prima formă, cea în *cod binar*, este dată în tabelul 6.9 şi a fost obţinută pornind de la definirea şi codificarea setului de instrucţiuni (Definiţia 6.9, figura 6.49 şi tabelele 6.7). Reprezentarea este foarte "concretă", oferind direct codurile binare, dar este, în acelaşi timp şi în primul rînd, foarte obscură. Manipularea unor

programe chiar foarte simple este practic imposibilă sub această formă. Această formă este direct "manipulată" de mașină.

| Adresă | Cod binar | Comentariu |
|---|---|---|
| 0010 | 0110 . xxxx . xxxx . 0000 | CONST ..., R0 |
| 0011 | 0000 . 0000 . 0001 . 1000 | $18_H$ |
| 0012 | 0101 . 0000 . xxxx . 0001 | LOAD R0, R1 |
| 0013 | 1101 . 0000 . xxxx . 0000 | INC R0, R0 |
| 0014 | 0101 . 0000 . xxxx . 0010 | LOAD R0, R2 |
| 0015 | 1011 . 0001 . 0010 . 0010 | PLUS R1, R2, R2 |
| 0016 | 1101 . 0000 . xxxx . 0000 | INC R0, R0 |
| 0017 | 0100 . 0000 . 0010 . xxxx | STORE R2, R0 |
| 0018 | 0000 . 0000 . 0101 . 0110 | 86 |
| 0019 | 0000 . 0000 . 0000 . 0101 | 5 |
| 001A | xxxx . xxxx . xxxx . xxxx | dest. rezultat |

**Tabelul 6.9**

| Adresă | Cod hexa |
|---|---|
| 0010 | 6xx0 |
| 0011 | 0018 |
| 0012 | 50x1 |
| 0013 | D0x0 |
| 0014 | 50x2 |
| 0015 | B122 |
| 0016 | D0x0 |
| 0017 | 402x |
| 0018 | 0056 |
| 0019 | 0005 |
| 001A | xxxx |

**Tabelul 6.10**

O formă mai compactă este cea a reprezentării în *hexazecimal* (uneori în *octal*) a codului instrucțiunilor. Programul nostru apare într-o astfel de formă în tabelul 6.10. În această formă încep să transpară unele din codurile asociate mnemonicelor fără ca programul să capete expresivitate pentru cine-l citește.

*Codul simbolic* începe să ofere o imagine inteligibilă programului, după cum se poate observa în continuare.

```
0010  CONST #18, R0     / Încarcă în R0 adresa primului operand /
0012  LOAD R0, R1       / Încarcă primul operand în R1/
0013  INC R0, R0        / Calculează adresa opereandului secund/
0014  LOAD R0, R2       / Încarcă operandul secund în R2/
0015  PLUS R1, R2, R2   / Adunarea propriuzisă/
0016  INC R0, R0        / Calculează adresa destinației/
0017  STORE R2, R0      / Stochează rezultatul/
0018  86                / Valoarea primului operand/
0019  5                 / Valoarea operandului secund/
001A  X                 / Locație liberă pentru rezultat/
```

Dificultățile imediate care mai rămîn în cazul codului simbolic sînt legate de gestionarea adreselor operanzilor ce nu întotdeauna pot fi cunoscute în timpul scrierii programelor. *Limbajul de asamblare* are avantajul esențial de a lucra cu adrese simbolice. Putem specifica printr-un nume locația de memorie cu care operăm, urmînd să asociem numelui o semnificație concretă, o adresă, ulterior, cînd acest lucru devine posibil. De asemenea, în limbajul de asamblare se mai pot da



diferite ordine care ajută la poziționarea programului în memorie, sau la definirea unor mărimi, etc.

Scrierea în limbaj de asamblare ne scutește de o serie de preocupări legate de poziționarea programului în memorie, de gestionarea explicită a unor locații de memorie cu care lucrăm. Programul nostru va fi rescris după cum urmează:

```
        ORG 16          / Definim începutul programului la adresa #10/
        CONST A, R0     / A reprezintă o etichetă ulterior definită/
        LOAD R0, R1
        INC R0, R0
        LOAD R0, R2
        PLUS R1, R2, R2
        INC R0, R0
        STORE R2, R0
A       DEC 86
        DEC 5
        DEC 0
        END
```

În programul anterior apar și o serie de *pseudoinstrucțiuni* (*ORG N, DEC N, END*) care, împreună cu instrucțiunile, fac parte din limbajul de asamblare definit după cum urmează:

*<linie program asamblare>* ::= *<etichetă><operație><comentariu>*

*<operație>* ::= *<Instrucțiune>* | *<Pseudoinstrucțiune>*

*<Instrucțiune>* ::= (conform *Definiția 6.9*)

*<Pseudoinstrucțiune>* ::= *ORG<N>* | *DEC<N>* | *HEX<N>* | *END*

*<N>* ::= *0* | *1* | *2* | ... | $2^{16}$-*1*

*<etichetă>* ::= *orice secvență de maximum șase simboluri*

*<comentariu>* ::= *text ce permite documentarea programului.*

Limbajul de asamblare este deja o formă în care se pot scrie programe de complexitate mai mare; el este, de asemenea, limbajul în care se pot optimiza puternic anumite programe cu condiția unei complexități limitate. Față de un limbaj de nivel înalt, prezintă avantajul că reflectă modul de funcționare al mașinii într-o manieră încă suficient de clară. În acest limbaj nu se mai manipulează explicit coduri sau adrese.

Într-un *limbaj de nivel înalt* mașina nu se mai vede; dispare orice urmă referitoare la locații de memorie, la registre sau alte resurse concrete ale sistemului. Dar, exprimarea va fi mult mai sugestivă pentru utilizatorul uman. Spre exemplu, același program într-un limbaj de nivel înalt, să zicem FORTRAN, arată în modul următor:

```
INTEGER A, B, C
DATA A, B /86, 5/
C = A + B
END
```

Productivitatea programării în limbaje de nivel înalt este incomparabil mai mare decît în limbaje de asamblare. De altfel, cele două raţiuni ale acestor categorii de limbaje sînt *independenţa de maşină* şi *productivitatea*.

Cum se trece de la un limbaj de asamblare sau unul de nivel înalt la codul maşină ce se execută? Pentru aceasta există programe dedicate.

*Asamblorul* este un program care translatează un program scris în limbaj de asamblare în limbaj cod maşină (cod binar). Acest program generează un cod univoc asociat programului de asamblat. El este asociat fiecărui tip de maşină în parte.

*Compilatorul* generează cod executabil pornind de la un limbaj de nivel înalt. Pe o maşină vor rula mai multe compilatoare pentru ca utilizatorul să poată avea acces la mai multe limbaje de nivel înalt. Dar chiar aceluiaşi limbaj, pe aceeaşi maşină, i se pot asocia mai multe compilatoare în funcţie de eficienţa dorită şi de resursele sau timpul disponibile. Un compilator poate genera un cod mai bun decît altul pentru acelaşi program şi pe aceeaşi maşină, în funcţie de dibăcia cu care a fost conceput sau în funcţie de resursele puse în joc. Realizarea unui bun compilator, cu toate mijloacele formale de care dispunem, este în mai mare măsură o artă decît o tehnică.

În limbaj de asamblare putem realiza, dacă stăpînim arhitectura maşinii, *programe mai eficiente*, pe cînd în limbaje de nivel înalt *programăm mai eficient*. Din acest motiv, de multe ori, realizăm compromisul ca porţiuni critice din programe de sistem să le "compilăm cu mîna", adică să le scriem în limbaj de asamblare. Eficienţa programelor şi programabilitatea sînt greu conciliabile.

### 6.6.4 Logica programată

Sistemele cu *logică programată* sînt, de regulă, microcalculatoare ($\mu C$) realizate centrat pe un microprocesor, eventual într-o tehnologie ce permite realizarea întregului sistem pe un singur cip, care *realizează funcţiuni inferioare celor realizate prin sistemele de ordinul trei*. În figura 6.59 este reprezentată o structură ce poate realiza funcţiuni logice programate. Teoretic vorbind un astfel de sistem, *dacă timpul nu este o variabilă restrictivă*, poate realiza programat funcţia *oricărui* circuit logic combinaţional cu *16* intrări şi *16* ieşiri, în sensul că poate asocia tuturor configuraţiilor binare de intrare $I_{15} \dots I_0$, cîte o configuraţie binară la ieşirile $O_{15} \dots O_0$.

**Definiţia 6.10** Un microcalculator cu intrările $I_n \dots I_0$ şi ieşirile $O_n \dots O_0$ (conform figurii 6.59) funcţionează ca un sistem cu logică programată dacă execută procedura *LOGICA PROGRAMATĂ* dată în procedura 6.12. □



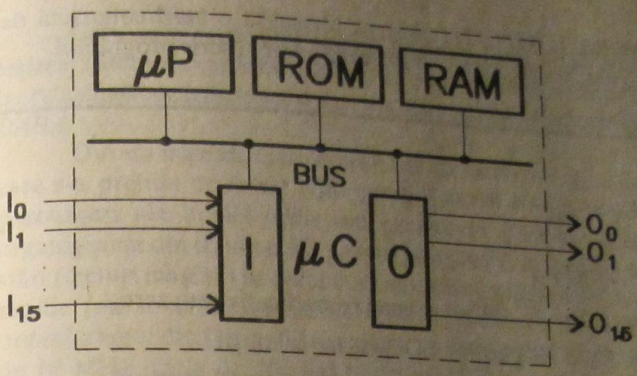



Figura 6.59

**Procedure** *LOGICĂ PROGRAMATĂ*

**loop** *Ri ← input* / Un registru oarecare se încarcă cu $I_{15} \dots I_0$./

*FUNCŢIE DIGITALĂ* / Se calculează în *Rj* o funcţie de valoarea din *Ri*./

*Out ← Rj* / Se încarcă valoarea din *Rj* la ieşirile $O_{15} \dots O_0$, care păstrează această valoare pînă la o nouă încărcare./

**repeat**

**end** *LOGICĂ PROGRAMATĂ*

**Procedura 6.12**

Cu o întîrziere dată de fiecare parcurgere a buclei infinite, ieşirile urmăresc evoluţia intrării prin intermediul funcţiei de transfer *FUNCŢIE DIGITALĂ* (procedura 6.12). Dacă procedura *FUNCŢIE DIGITALĂ* depinde nu numai de variabila din $R_i$, ci şi de una "ascunsă", atunci se pot realiza şi funcţiuni ce corespund unor sisteme de ordin superior lui zero. O astfel de variabilă "ascunsă" ar putea fi una de stare şi atunci sistemul funcţionează ca un automat finit cu intrările $I_i$ şi ieşirile $O_j$ ($i, j = 0, 1, \dots, n$).

**Exemplul 6.25** Se cere realizarea unui sistem care, avînd la intrare numărul *N*, reprezentat ca un întreg pozitiv pe *16* biţi, să genereze pe ieşire mărimea

$$2^{i+1} - 1$$

dacă

$$\lfloor \log_2 N \rfloor = 2^i,$$

pentru $N>0$, altfel ieşirea va lua valoarea zero.

Această funcţie poate fi uşor realizată cu un *CLC* care primeşte la intrare o configuraţie binară de *16* biţi şi generează pe ieşire o alta care are *1* pe toate poziţiile de ordin binar egal sau mai mic decît ordinul binar cel mai semnificativ de la intrare, ce are valoarea *1*. Spre exemplu, dacă $I_{15} \dots I_0 = 0000.0101.0001.0110$,

atunci $O_{15} \dots O_0 = 0000.0111.1111.1111$. Ieşirea formează o *bară* compactă de unu-uri. Procedura 6.13 descrie algoritmul după care poate fi scris programul.

```
Procedure BARĂ
   loop   R0 ← Input
          if     R0 ≠ 0
            then R1 ← FFFF_H
                 while R0 < 2^15 / Bitul cel mai semnificativ este zero./
                   do  R0 ← R0 plus R0  / Deplasare la stînga./
                       R1 ← SHR R1  / Deplasare la dreapta (divizare)./
                 repeat
            else R1 ← 0000_H
          endif
          Out ← R1
   repeat
end BARĂ                                          Procedura 6.13
```

În limbaj de asamblare rezultă următorul program (adresa echipamentului de *I/O* este *#100*, iar programul va fi implantat în memorie începînd cu adresa *#200*):

```
                ORG   #200
LOOP            CONST  #100, R2
                LOAD   R2, R0
                IF ZERO JMP NUL
                CONST  #FFFF, R1
                CONST  #8000, R3
SUS             AND  R0, R3, R4
                IF NOT ZERO JMP GATA
                PLUS  R0, R0, R0
                SHR  R1, R1
                JMP  SUS
GATA            STORE  R1, R2
                JMP  LOOP
NUL             CONST  #0, R1
                JMP  GATA
                END
```



Care este intervalul minim de timp între două comutări ale $I_i$ pentru ca ieşirea sistemului anterior definit să poată urmări variaţia? Vom putea afla acest lucru calculînd cît durează o parcurgere maximă a buclei infinite *loop* din procedura *BARĂ*.

Durata maximă între două citiri ale intrărilor $I_{15} \dots I_0$ este dată de situaţia în care s-a preluat de la ele configuraţia binară $0001_H$, deoarece vor fi realizate *15* parcurgeri ale buclei *while* din procedura *BARĂ*. Conform descrierii dată în organigrama din figura 6.52, pentru $\mu P_0$ (cu *PC* în *RALU*) rezultă un număr de *272* stări (cicluri maşină) în cazul maxim, dacă memoria internă şi interfeţele sînt foarte rapide (nu se activează semnalul *WAIT*). Pentru $\mu P_1$ (cu *PC* distinct), conform organigramei din figura 6.56, rezultă un număr de *188* cicluri. Deci, pentru un ceas de *10* MHz (ciclu de *100* ns) obţinem *27,2* $\mu$s, pentru $\mu P_0$ şi *18,8* $\mu$s pentru $\mu P_1$. Frecvenţa maximă la care poate comuta intrarea pentru a putea fi urmărită fidel va fi de *36,7* KHz, pentru sistemul cu $\mu P_0$ şi de *53,1* KHz, pentru sistemul realizat cu $\mu P_1$.

*Atenţie*! Ce probleme pune preluarea, fatalmente asincronă, a configuraţiei binare $I_{15} \dots I_0$? Nu pune problemele obişnuite datorită naturii funcţiei de transfer. De ce? Cititorul va căuta singur răspunsul. □

Tehnica logicii programate permite şi realizarea unor funcţii asociate sistemelor de ordinul unu (sistemelor de memorare). Un exemplu pe care-l poate dezvolta cititorul este cel al unei memorii *FIFO*. Vom prezenta, în continuare, pe larg realizarea unei funcţiuni de automat finit (asociate sistemelor de ordinul doi) cu ajutorul unui $\mu$C utilizînd logica programată.

**Exemplul 6.26** Tot prin logică programată poate fi realizat şi un sistem care să recepţioneze şiruri de numere pe intrarea $I_{15} \dots I_0$ şi să genereze la ieşire tot şiruri de numere după o funcţie de tipul celei din figura 6.60. Caracteristica de transfer este de tip *histerezis*, definită prin întregii *A, B, C* şi *D* ce îndeplinesc următoarele condiţii:

$$A, B, C, D \in [0, 2^{16}),$$
$$A < B,$$
$$C < D.$$

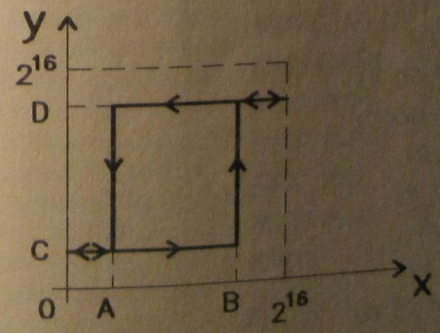



Figura 6.60

Procedura *HISTEREZIS* (procedura 6.14) defineşte funcţia de transfer a sistemului cu logică programată. Observăm că este definită o funcţie de tip *automat finit* cu registrul de stare $R_1$. În $R_1$ se află o configuraţie binară de tip variabilă de stare, deoarece la fiecare parcurgere a buclei infinite valoarea anterioară poate fi luată în considerare.

---

**Procedure** *HISTEREZIS* / Pe $x$, $A < B$ şi pe $y$, $C < D$./

```
loop R0 ← Input
     if    R0 ≤ A
           then  R1 ← C
           else  if R0 ≥ B
                    then R1 ← D
                 endif
     endif
     Out ← R1
repeat
end HISTEREZIS
```

**Procedura 6.14**

---

Am putea privi sistemul ca un automat definit prin

$$A_H = (X, Y, Q, \lambda, \delta)$$

unde: $X \in [0, 2^{16})$, $X$ întreg; $Q = \{C, D\}$, $Y \equiv Q$, iar funcţiile $\lambda$ şi $\delta$ definite prin procedura *HISTEREZIS*.

Programul scris în limbaj de asamblare va fi următorul:

```
        ORG #300
LOOP    CONST #100, R2
        LOAD R2, R0
        CONST A, R3
        MINUS R3, R0, R3
        IF CR JMP DOWN
        CONST C, R1
GATA    STORE R1, R2
        JMP LOOP
DOWN    CONST B, R3
        MINUS R0, R3, R3
        IF CR JMP GATA
        CONST D, R1
        JMP GATA
A       HEX #150
B       HEX #A00
C       HEX #80
D       HEX #3AB
        END
```

Am scris acest program respectînd strict Definiţia 6.10. El poate fi optimizat dacă este introdusă o zonă iniţială de "amorsare", în care se pregătesc anumite valori în registrele interne. Vom folosi un număr mai mare de registre, dar bucla infinită *loop* va fi parcursă într-un timp mai scurt, asigurînd o viteză de reacţie mai mare sistemului. Varianta secundă este următoarea:

```
        OR6 #300
        CONST #100, R2
        CONST A, R3
        CONST B, R4
        CONST C, R5
        CONST D, R6
LOOP    LOAD R2, R0
        MINUS R3, R0, R7
        IF CR JMP NU
        MOVE R5, R1
GATA    STORE R1, R2
        JMP LOOP
NU      MINUS R0, R4, R7
        IF CR JMP GATA
        MOVE R6, R1
        JMP GATA
A       HEX #150
B       HEX #A00
C       HEX #80
D       HEX #3AB
        END
```

Cititorul va face efortul de a calcula frecvenţa maximă la care poate funcţiona sistemul cu cele două programe. Va rezulta avantajul net al celei de a doua variante. □

Experienţa exemplului anterior impune redefinirea procedurii *LOGICĂ PROGRAMATĂ* sub forma procedurii 6.15. Definiţia 6.10 va fi amendată în consecinţă.

---

**Procedure** *LOGICĂ PROGRAMATĂ*
  *DEFINIRE CONTEXT*
  **loop** *Ri* ← *Input*
    *FUNCŢIE DIGITALĂ*
    *Out* ← *Rj*
  **repeat**
**end** *LOGICĂ PROGRAMATĂ*

**Procedura 6.15**

---

Există foarte multe aplicaţii la care timpul de reacţie al sistemului digital nu se impune la valori foarte mici. În domenii mecanice, în domeniul automatizărilor şi în altele, timpi de reacţie de zeci de microsecunde, sau mai mari, sînt perfect tolerabili. În aceste situatii logica programată oferă o soluţie de maximă flexibilitate. Este semnificativ în acest sens ultimul exemplu în care parametrii funcţiei de transfer de tip histerezis puteau fi setaţi ca valori, obţinîndu-se *orice* caracteristică fără implicaţii structurale.

O imensă clasă de aplicaţii ale microprocesoarelor, unde acestea-şi dovedesc din plin utilitatea, o constituie sistemele cu logică programată.

### 6.6.5 Concluzii

Pe măsură ce structurile de procesoare descrise în acest subcapitol s-au complicat, funcţionalitatea lor a început să depindă din ce în ce mai puţin de structura fizică, trecînd sub controlul unor structuri simbolice. Ne interesează din ce în ce mai puţin ordinul unui sistem ce conţine un procesor, deoarece funcţia ansamblului este din ce în ce mai mult dată de forma particulară a unei structuri simbolice. Această situaţie culminează cu definirea funcţiei unui $\mu C$, centrat pe un $\mu P$, prin intermediul unui *program* scris în limbaj de asamblare. Care este ordinul structurii din figura 6.59? Cu siguranţă este superior lui trei. Cititorul poate determina acest lucru. Dar în etapa în care am ajuns, acest lucru nu mai este esenţial deoarece controlul funcţional scapă de sub incidenţa strictă a structurii fizice. Exemplele date sau sugerate în paragraful anterior arată că *aceeaşi structură fizică diferit programată* poate îndeplini funcţiuni net distincte.



*Flexibilitatea funcţională* maximă astfel obţinută este plătită cu preţul unor *performanţe în viteză drastic atenuate*. De multe ori aceste performanţe sînt acceptabile şi astfel logica programată capătă un rost în lumea procedeelor de implementare a unor funcţii digitale.

● **Evoluţia sistemelor de procesare** este marcată prin două tendinţe:

- realizarea funcţiei de control prin structurarea suplimentară a automatului (*AC* evoluează către structura de *CROM*);

- introducerea unor structuri efectorii suplimentare (pe lîngă sau în *RALU*) orientate către funcţii critice ale sistemului (spre exemplu, un înmulţitor combinaţional, eventual în virgulă mobilă).

Pe această cale vom ajunge, în subcapitolul următor, la procesoarele *microprogramate*. În etapa pe care o parcurgem şi în cea următoare progresele tehnologice permit realizarea pe un cip a unor structuri din ce în ce mai complexe, astfel încît noţiunea distinctă de *microprocesor* îşi va pierde sensul. Orice procesor va putea fi realizat pe un singur cip şi diferenţele, încă existente, între calculatoarele de mare putere şi cele centrate pe microprocesoarele tipice, vor dispare.

## 6.7. Procesoare microprogramate

Microprocesoarele sînt procesoare la care funcţia de control intern este minimizată, în sensul că automatul de control, *AC*, este gîndit ca foarte simplu. *Procesoarele microprogramate* se situează la extrema în care *funcţia de control intern are o pondere foarte importantă*. Dacă un microprocesor foloseşte cît mai direct codul instrucţiunii (biţii din *RI* comandă direct funcţionarea *RALU*), în cazul imensei majorităţi a instrucţiunilor, la o structură microprogramată, codul instrucţiunii acţionează cu preponderenţă asupra automatului de control, declanşînd *secvenţe complexe de comandă*. Să ne amintim că numai pentru cîteva instrucţiuni (cele de salt) microprocesorul folosea majoritatea stărilor automatului de control; în acest caz toţi biţii utili ai instrucţiunii erau interpretaţi numai de acest automat. Această caracteristică netipică a microprocesorului este dezvoltată în gîndirea structurilor microprogramate, impunînd amplificarea rolului automatului de control ce apare, în acest caz, sub forma unei unităţi de tip *CROM* (vezi paragraful 5.6.2).

Structura minimală a unei maşini microprogramate este reprezentată în figura 6.61, unde *RALU* poate fi de tipul celor din figurile 5.97 sau 5.98, iar *CROM* de tipul celui din figura 5.87.

Arhitectura unui microprocesor este explicit definită prin setul de instrucţiuni. O maşină microprogramată poate fi concret realizată pentru una dintr-o mulţime de arhitecturi, în funcţie de particularizarea conţinutului *ROM*-ului din *CROM*. La un microprocesor arhitectura precede structura, pe cînd la un procesor microprogramat arhitectura urmează definirii structurii. Procesul poate fi iterativ

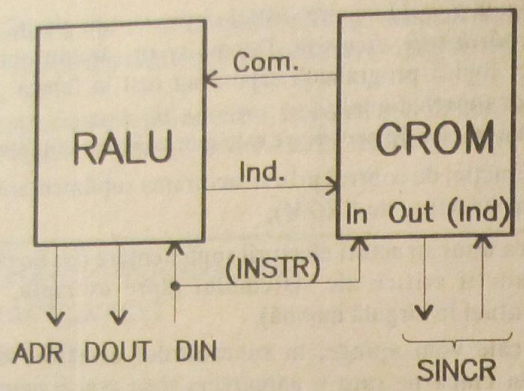



**Figura 6.61**

în fazele inițiale ale proiectării unei mașini microprogramate, în sensul că și structura poate suferi redefiniri, măcar parțiale, ca urmare a unei tentative nereușite de implementare performantă a unei arhitecturi. Deci *microprocesorul este predefinit funcțional* iar *procesorul microprogramat este postdefinit funcțional* în raport cu structura fizică.

*Mobilitatea arhitecturală* a structurilor microprogramate permite *creșterea performanței* prin *adecvare funcțională* la problema ce trebuie rezolvată. De multe ori structurile de acest tip nu se constituie în mașini universale, modelîndu-și funcțiile (setul de instrucțiuni) conform aplicației concrete. Astfel, performanța în rezolvarea unor probleme concrete poate crește sensibil.

Microprocesoarele din ultimele generații tind către structura mașinilor microprogramate, denumirea păstrînd-o numai din considerente comerciale. Majoritatea microprocesoarelor de *32* de biți sînt de fapt procesoare microprogramate realizate pe un singur cip.

Dacă în cazul microprocesoarelor am început prin a defini o *arhitectură*, în cazul mașinilor microprogramate vom începe cu definirea *limbajului de microprogramare*, definind prin aceasta *o clasă de arhitecturi* posibile.

### 6.7.1 Limbaj de microprogramare

Actualizarea funcției *CROM*-ului se face prin definirea conținutului *ROM*-ului. Deoarece, de regulă, imaginea din această memorie este foarte complexă, se impune folosirea unui limbaj în care descrierea conținutului *ROM*-ului să poată fi făcută într-o manieră cît mai flexibilă și controlabilă. Acest rol este îndeplinit de *limbajul de microprogramare* în care se descriu *comenzile* și *modul lor de înlănțuire*.

Limbajul de microprogramare permite definirea *microprogramelor*, a secvențelor de comenzi sau *microinstrucțiuni* ce sînt asociate unei *instrucțiuni*.



**Definiția 6.11**

*<Microinstrucțiune>* ::= *<Etichetă><Com.RALU><Com. ext.><Mod><Test><Branch>*,

*<Etichetă>* ::= *orice secvență de maximum șase caractere, ce nu va fi folosită în această definiție*,

*<Com.RALU>* ::= *<LA><RA><SL><SR><CONST><FUNC><WR>*,

*<Com.ext.>* ::= *READ* | *WRITE* | *INTA* | −,

*<Mod>* ::= *RST* | *INIT* | *JMP* | *CJMP* | −,

*<Test>* ::= *CR* | *OVFL* | *ZERO* | *EQAL* | *PAR* | *SGN* | *WAIT* | | *INT* | −,

*<Branch>* ::= *<Etichetă>*,

*<LA>* ::= *L0* | *L1* | *L2* | ... | *LF* | −,

*<RA>* ::= *R0* | *R1* | *R2* | ... | *RF* | −,

*<SL>* ::= *DIN* | −,

*<SR>* ::= *K* | −,

*<CONST>* ::= *#0* | *#1* | ... | *#FFFF* | −,

*<FUNC>* ::= *AND* | *OR* | *XOR* | *LEFT* | *RIGHT* | *PLUS* | *MIN* | *SHR* | −,

*<WR>* ::= *W* | −.

În cîmpul *<Mod>* valoarea implicită este *INC*. Cîmpul *<SR>* nu va fi folosit deoarece este redundant, datorită valorii din cîmpul *<CONST>*; dacă apare o constantă în microinstrucțiune este implicită valoarea *K*, ce va fi automat generată de translatorul de microprograme. □

Mașina la care se referă acest limbaj de microprogramare este reprezentată în figura 6.62. Funcționarea este evidentă pentru că subansamblurile ei, *CROM* și *RALU*, au fost descrise în secțiuni anterioare. Vom da în continuare cîteva exemple de microprograme.

A rezultat un cuvînt de microprogram de *49* biți. Prin codificarea cu doi biți a *<Com.ext.>* se poate economisi un bit reducîndu-se dimensiunea la *48* biți - un multiplu de *4*, ușor de implementat eficient cu *ROM*-uri de *4* sau *8* biți pe cuvînt. În consecință în schemă va trebui adăugat un decodificator pentru cîmpul *<Com.ext.>*. Rezultă structura cuvîntului de microprogram din figura 6.63. Codificarea fiecărui cîmp o poate ușor imagina cititorul și singur.

**Exemplul 6.27** Microprogramul de *FETCH* este cel care asigură bucla infinită în care funcționează procesorul microprogramat, imediat după acționarea semnalului *RESET*, care șterge registrul *R* din *CROM* forțînd executarea microinstrucțiunii de la adresa zero. (Semnalul *RESET* trebuie sincronizat cu *CK* pentru a asigura pornirea corectă a mașinii.) Vom considera, similar cazului microprocesorului, că registrul *F* este folosit drept *PC* (program counter) iar registrul *E* drept *SP* (stack pointer). Acest microprogram este implantat în memoria *ROM* începînd de la adresa zero.

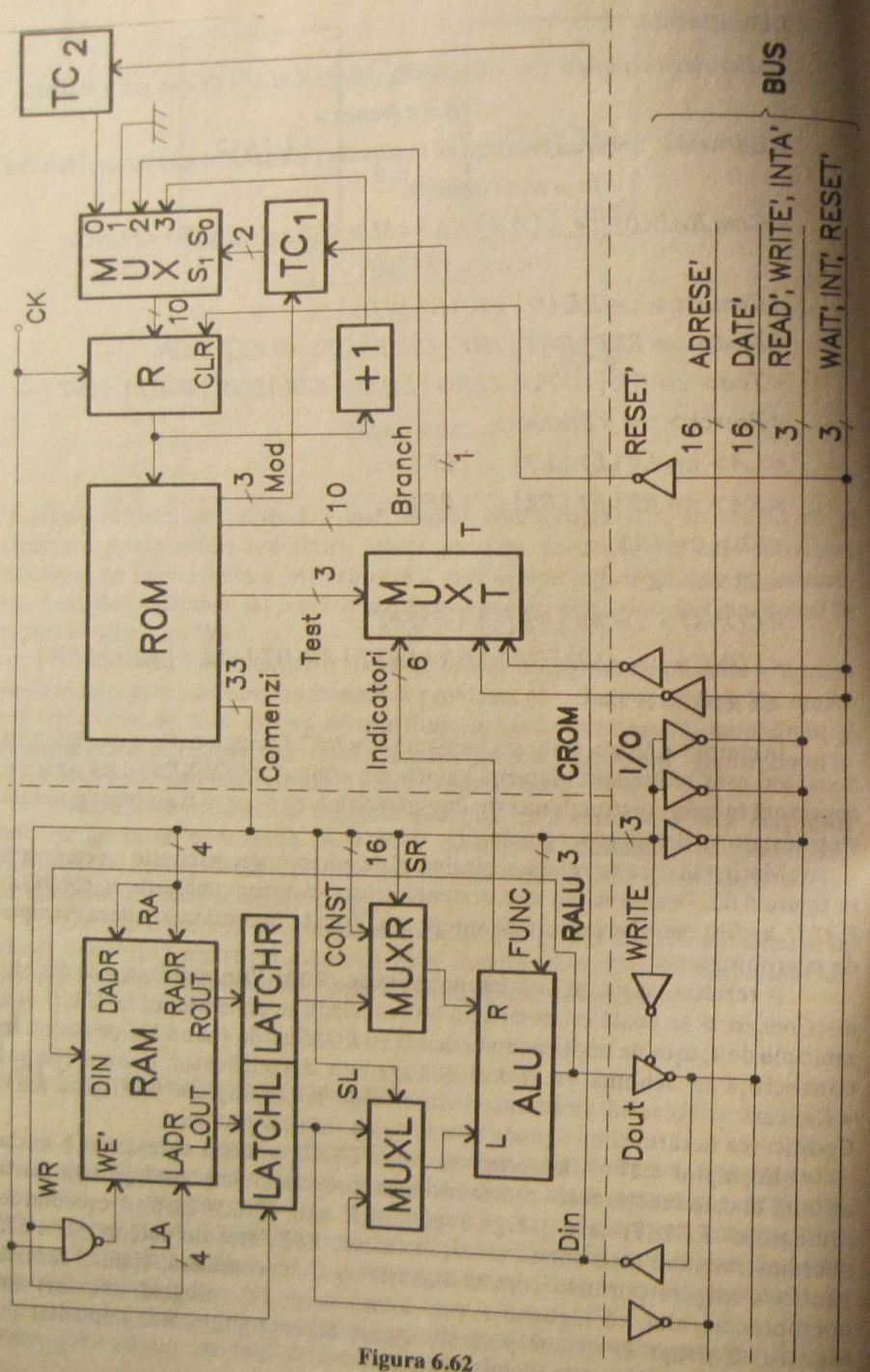



Figura 6.62



| 2 | 2 | 1 | 1 | 16 | 3 | 1 | 2 | 3 | 3 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| LA | RA | SL | SR | CONST | FUNC | WRITE | Com ext | | Mod | Test | Branch |

Figura 6.63

```
000             LF RF XOR W / Resetarea registrului F, PC = zero./
001     FETCH   CJMP INT INTER / Salt condiţionat la locaţia INTER
                  dacă există semnalul de întrerupere, INT./
002     LOOP    READ LF CJMP WAIT LOOP / Comandă de citire de la
                  adresa din registrul F (PC-ul) atît timp cît este activat
                  semnalul WAIT./
003             READ LF INIT RF #1 PLUS W / Codul citit de la adresa
                  din F (PC) este folosit pentru a iniţializa CROM-ul.
                  Conţinutul registrului F este sumat cu constanta #1 ./
004     INTER   WRITE LE RF RIGHT CJMP WAIT INTER / Scrie la adresa
                  din registrul E (SP) valoarea PC-ului (salvează PC-ul
                  în stivă)./
005             LE RE PLUS #1 W / Increment SP./
006             #100 RIGHT RF W JMP LOOP / Încarcă în PC adresa (în
                  acest caz #100) de la care începe în RAM-ul exterior
                  programul de tratare al întreruperii./
```

Secvenţa anterioară de microinstrucţiuni este minimală, asigurînd o funcţionare corectă a maşinii microprogramate. Ea mai poate fi, eventual, complicată în zona tratării întreruperilor şi ar fi bine dacă s-ar putea simplifica în zona de *FETCH* propriuzis. Orice microprogram la care se sare prin *INIT*, în microinstrucţiunea *003*, va trebui să se termine cu o linie de microprogram care să conţină mnemonicele *JMP INIT*, declanşînd aducerea unei alte instrucţiuni din memoria externă procesorului. □

Un procesor microprogramat poate avea instrucţiuni de la cele mai simple la cele mai complexe, singura restricţie, importantă, este cea dată de capacitatea memoriei *ROM* de microprogram. Ceea ce într-un microcalculator realizat cu microprocesor se realizează printr-un program, poate avea asociat într-un procesor microprogramat o singură instrucţiune (un microprogram). Să reluăm, în acest sens, funcţia executată de programul folosit drept exemplu în paragraful 6.6.3.

**Exemplul 6.28** Se consideră instrucţiunea care adună, modulo $2^{16}$, două numere întregi şi pozitive. Adresa primului operand se află în *RAM* la adresa *PC+1*, dacă la *PC* se află codul instrucţiunii. Formatul acestei instrucţiuni este reprezentat în figura 6.64. La adresele imediat următoare celei la care se află primul operand se află cea a celui de al doilea operand şi cea în care depunem rezultatul.

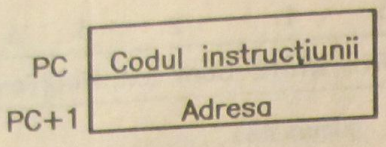



**Figura 6.64**

Rezultă următorul microprogram:

*UNU* *READ LF CJMP WAIT UNU R0 DIN LEFT W* / Se citeşte în *R0* adresa primului operand de la *PC*, care a fost incrememtat în microinstrucţiunea *003* din exemplul anterior./

*LF RF #1 PLUS W* / Increment *PC* pentru ca în rutina *FETCH* să ţintească corect instrucţiunea următoare./

*DOI* *READ L0 CJMP WAIT DOI R1 DIN LEFT W* / Operandul prim este citit în registrul *1*./

*L0 R0 #1 PLUS W* / Incrementarea adresei./

*TREI* *READ L0 CJMP WAIT TREI* / Se comandă citirea celui de al doilea operand şi se aşteaptă pînă cînd citirea este efectuată (*WAIT = 0*)./

*READ L0 DIN R1 PLUS W* / *Adunarea propriuzisă*./

*L0 R0 #1 PLUS W* / Calculul adresei destinaţiei rezultatului./

*PATRU* *WRITE L0 R1 RIGHT CJMP WAIT PATRU* / Scrierea rezultatului din *R1* la adresa din *R0*./

*JMP FETCH* / Salt la rutina prin care se aduce instrucţiunea următoare (*Exemplul 6.27*)./

Nu am numerotat liniile acestui microprogram deoarece el va fi implantat în *ROM* începînd cu o adresă care, în acest moment, nu este esenţială. La această adresă se va iniţializa *CROM*-ul folosind comanda *INIT* şi *Codul Instrucţiunii* citit de pe calea de date, drept urmare a comenzii de citire cu adresa din *PC* (registrul *F*).

Programul din paragraful 6.6.5 care realizează aceeaşi acţiune se desfăşoară pe o durată de *22* cicluri (dacă *WAIT = 0* tot timpul, lucrînd cu o memorie rapidă) pe $\mu P_0$ (cel cu *PC*-ul în *RALU*). Pe $\mu P_1$ (cu *PC* realizat distinct) acelaşi program durează *15* cicluri. Pe maşina microprogramată, în aceleaşi condiţii (*WAIT = 0*), aceeaşi funcţie se realizează în *11* cicluri maşină, deci de două ori mai repede decît pe o maşină similară (fără *PC* structurat distinct). Avantajul este evident. □

Viteza de execuţie a unor funcţii microprogramate faţă de cazul în care sînt programate este întotdeauna mai mare, în condiţii tehnologice echivalente, în primul rînd datorită faptului că se evită ciclurile de *FETCH* ale instrucţiunilor.



### 6.7.2 Variante structurale ale procesoarelor microprogramate

Modelul prezentat în paragraful anterior este unul minimal, util pentru expunerea ideilor de bază referitoare la o maşină microprogramată. Configuraţiile practic realizate sînt dotate cu facilităţi suplimentare ce permit creşterea performanţelor de viteză sau programabilitate.

● **Testarea condiţiei negate** pentru salt este uneori o facilitate care permite scrierea unor microprograme mai performante sau măcar mai clare. (Claritatea unui microprogram nu este o calitate neglijabilă atunci cînd complexitatea lui depăşeşte o anumită limită. Deci atenţie!) Pentru introducerea acestei facilităţi definiţia liniei de microprogram se modifică după cum urmează:

**Definiţia 6.12** În *Definiţia 6.11* facem următoarea modificare:

*<Microinstrucţiune>* ::= *<Etichetă><Com.RALU><Com.ext.><Mod>* [*NOT*]*<Test><Branch>*,

unde simbolurile dintre parantezele drepte pot apare sau nu. □

Structura maşinii se completează (figura 6.65) cu un *XOR* la ieşirea *MUX T*, comandat de un bit din *ROM*. Cu acest circuit se poate nega comandat valoarea bitului de condiţie testat.

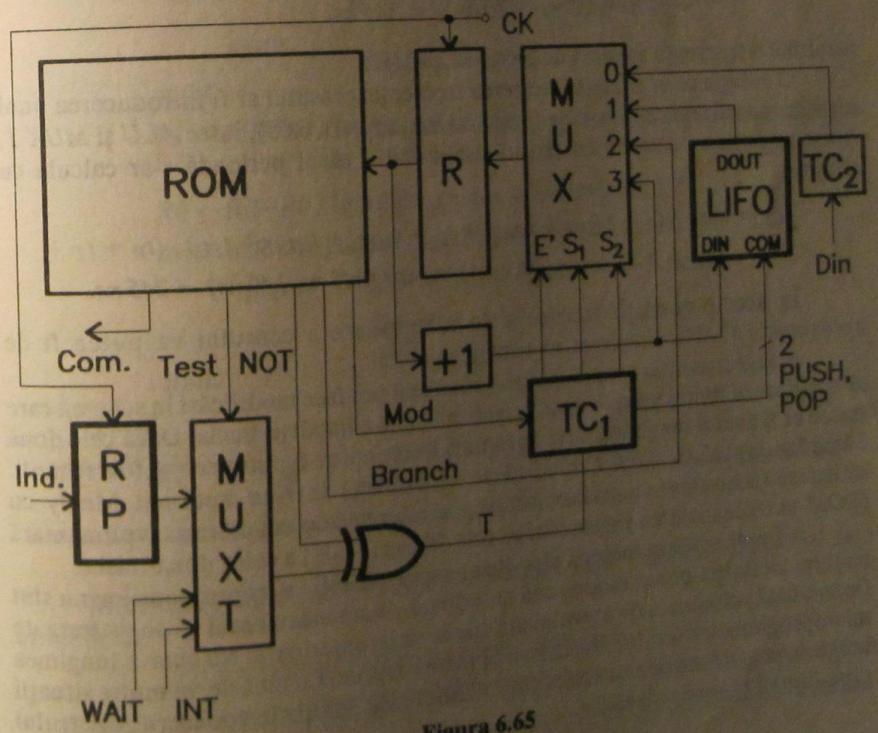



**Figura 6.65**

• **Introducerea unui registru pipe-line între RALU şi CROM pe calea indicatorilor** este o soluţie ce permite creşterea frecvenţei ceasului cu care funcţionează procesorul. Bucla cea mai lungă ce se închide în această structură este cea care limitează frecvenţa maximă a tactului. Această buclă are următorul traseu:

- propagarea prin registrul *R* din *CROM* ($t_R$),
- propagarea prin memoria de microprogram ($t_{ROM}$),
- timpul de acces la latch-urile *RAM*-ului din *RALU* ($t_{RAM}$),
- timpul de încărcare al latch-urilor ($t_L$),
- timpul de propagare prin *MUX L* sau *MUX R* ($t_{MRL}$),
- timpul de propagare la ieşirea *ALU* a indicatorului cel mai leneş, care este *CR* sau *OVFL* ($t_{ALU}$),
- timpul de propagare prin *MUX T* ($t_T$),
- (- timpul de propagare prin *XOR*-ul ce neagă condiţia de test ($t_{XOR}$)),
- timpul de propagare prin $TC_1$ ($t_1$),
- timpul de propagare prin *MUX*, pe căile de selecţie ($t_{MUX}$),
- timpul de set-up la intrarea registrului *R* ($t_{SU}$).

Dacă dispunem de o tehnologie în care timpul mediu de propagare printr-o poartă este de *15* ns, spre exemplu, timpul total asociat buclei ($t_{BUCLĂ}$) va fi aproximativ următorul (valorile numerice apar în ordinea listei anterioare):

$$t_{BUCLĂ} = 35\ ns + 50\ ns + 50\ ns + 30\ ns + 15\ ns + 60\ ns + 15\ ns + 30\ ns + \\ + 15\ ns + 35\ ns + 10\ ns + = 345\ ns,$$

rezultînd o frecvenţă maximă de lucru de 2,9 MHz.

O soluţie ce ar permite creşterea frecvenţei ceasului ar fi introducerea unui registru, numit *registru pipe-line* (notat cu *RP* în figura 6.65), între *ALU* şi *MUX T*. Acest registru ar comuta cu ceasul sistemului a cărui perioadă s-ar calcula cu relaţia:

$$T_{CK} = max\ ((t_R + t_{ROM} + t_{RAM} + t_L + t_{MRL} + t_{ALU} + t_{SU})\ ,\ (t_R + t_T + \\ + t_{XOR} + t_1 + t_{MUX} + t_{SU})) = max\ (245\ ns,\ 140\ ns) = 245\ ns.$$

În aceste condiţii frecvenţa de funcţionare a ceasului va putea fi de aproximativ *4* MHz. Creşterea nu este neglijabilă!

Atragem atenţia asupra faptului că se mai pot face modificări în schemă care să echilibreze timpii pe cele două trasee în care s-a împărţit bucla. Dacă cele două trasee ar fi avut o lungime foarte apropiată frecvenţa ar fi putut creşte foarte mult. Sugerăm faptul că dacă *CROM*-ul se transformă într-un automat Mealy cu întîrziere (echivalent cu cel studiat) se poate produce o echilibrare suplimentară (*ROM*-ul trece înaintea registrului cu cele *50* ns ale sale) a celor două trasee.

• **Implicaţiile structurii pipe-line asupra scrierii microprogramelor** nu sînt majore. Definiţia poate rămîne cea anterioară cu amendamentul că se va testa de fiecare dată valoarea indicatorului din starea strict anterioară. Nu cumva lungimea microprogramelor creşte? Da, dar nu în toate cazurile. Cu dibăcie, în multe situaţii lungirea poate fi evitată sau minimizată astfel încît soluţia introducerii registrului pipe-line să rămînă valabilă.



Spre exemplu, microprogramul din *Exemplul 6.28* nu se modifică absolut de loc. Evident! Nu se testează nici un indicator de la *ALU*. Sînt multe astfel de microprograme. Vom încerca să construim şi un exemplu mai puţin avantajos, unul în care să se testeze cît mai mulţi indicatori la ieşirea din *ALU*.

**Exemplul 6.29** Vom scrie un microprogram ce realizează înmulţirea a două numere de *16* biţi (deînmulţitul în *R0* şi înmulţitorul în $R_1$) cu rezultatul pe *16* biţi stocaţi în registrul $R_2$. Algoritmul folosit este descris în procedura 6.16.

---

**Procedure** *PRODUS* / Înmulţirea întregilor pozitivi pe 16 biţi. La depăşire se forţează ca rezultat "infinitul" reprezentării. Deînmulţit : *R0* ; Înmulţitor : *R1*; Produs : *R2*. /

```
R2 ← zero / iniţializarea acumulatorului. /
while R1 ≠ 0
  do  if PAR=1 în R1 / PAR este bitul cel mai puţin semnificativ . /
        then R2 ← R2+R0 / Sumare parţială în acumulator. /
             if CR / Depăşirea capacităţii de reprezentare. /
               then   R2 ← #FFFF / Produsul ← " infinit ". /
                      R1 ← zero / Pentru a opri rularea buclei. /
             endif
      endif
      R1 ← SHR R1 / Deplasare la dreapta pentru inspectarea înmulţitoru-
                    lui. /
      R0 ← R0+R0 / Îmnulţire cu doi a deînmulţitului. /
      if    CR / Depăşirea capacităţii de reprezentare. /
            then    R2 ← #FFFF
                    R1 ← zero
      endif
  repeat
end PRODUS
```

**procedura 6.16**

---

Microprogramul scris pentru o maşină fără registrul pipe-line este următorul:

```
        L2 R2 XOR W  / Resetarea registrului R2 /
LOOP    L1 LEFT CJMP ZERO FETCH  / Dacă înmulţitorul este zero, atunci
                                   operaţia se încheie cu saltul la rutina FETCH prin care se va iniţia-
```

liza executarea următoarei instrucţiuni./

*L1 LEFT CJMP PAR SUMĂ*

*DEPL L0 R0 PLUS W CJMP CR ERROR* / Prin sumarea valorii obţinute prin deplasare se depăşeşte capacitatea de reprezentare./

*L1 R1 SHR W JMP LOOP*

*SUMĂ L0 R2 PLUS W CJMP NOT CR DEPL* / Acumulare în $R_2$./

*ERROR R2 #FFFF RIGHT W JMP FETCH* / Setarea registrului *R2* la valoarea *#FFFF* ("infinitul" reprezentării)./

Dacă procesorul este realizat cu registru pipe-line va rezulta următorul microprogram:

*L2 R2 XOR W*

*L1 LEFT*

*LOOP CJMP PAR SUMA L1 R1 SHR W*

*SHL CJMP ZERO FETCH L0 R0 PLUS W* / Se testează indicatorul *ZERO* rezultat din microinstrucţiunea anterioară./

*CJMP CR ERROR* / Se testează bitul de *CR* din operaţia anterior executată./

*JMP LOOP L1 LEFT* / Este trecut *R1* prin *ALU* pentru a se înregistra în registrul pipe-line indicatorii corespunzători valorii din acest registru, în vederea testării în ciclul maşină următor./

*SUMA L0 R2 PLUS W*

*CJMP NOT CR SHL*

*ERROR R2 #FFFF RIGHT W JMP FETCH.*

Cel de al doilea microprogram are două linii în plus, ocupînd cu *28,6 %* mai multe locaţii de memorie în *ROM*. Timpul *mediu* de execuţie al primului microprogram este de *74* cicluri maşină, iar al celui de al doilea de *82* cicluri, deci cu *10,8 %* mai multe cicluri. Dar, cît este timpul efectiv ţinînd cont de faptul că procesorul cu registru pipe-line poate funcţiona la o frecvenţă mai mare? Dacă vom lua în consideraţie estimările anterioare va rezulta că prima variantă de procesor va executa produsul în $74 \times 345\ ns = 25{,}53\ ms$, iar cea de a doua variantă va executa aceeaşi operaţie în $82 \times 245\ ns = 20{,}09\ ms$.

Efortul structural neglijabil presupus de introducerea registrului pipe-line este justificat prin această creştere de *27 %* a vitezei de execuţie (pentru acest exemplu!). □

Eficienţa introducerii registrului pipe-line depinde în ultimă instanţă de abilitatea proiectantului structurii, pentru început, iar mai apoi de eficacitatea celui ce scrie microprograme.



• **Introducerea unei stive în CROM** se justifică în funcţie de natura aplicaţiilor implementate pe structura microprogramată. Într-un set ampiu de funcţii complexe este foarte probabilă apariţia unor secvenţe de microinstrucţiuni care se repetă identic în cadrul mai multor microprograme sau chiar în acelaşi microprogram, în zone diferite. Folosirea mecanismului de subrutină (submicroprogram) se poate dovedi foarte utilă prin faptul că:

- se minimizează spaţiul ocupat în *ROM*-ul *CROM*-ului;

- o subrutină, odată scrisă şi verificată, poate fi folosită în scrierea mai eficientă a microprogramelor.

Deci, pentru stocarea adreselor de retur, va trebui să completăm structura *CROM*-ului cu o stivă şi cu mecanismele adecvate de comandă. Va rezulta configuraţia fizică de *CROM* din figura 6.65, în care am introdus şi celelalte modificări discutate în acest paragraf (*XOR*-ul pentru negarea condiţiei testate şi registrul pipe-line (*RP*)). De asemenea, definiţia limbajului de microprogramare se modifică.

**Definiţia 6.13** Limbajul de microprogramare modificat prin *Definiţia 6.12* se completează după cum urmează:

$$<Mod> ::= RST \mid INIT \mid JMP \mid CJMP \mid CALL \mid CCALL \mid RET \mid -$$

în continuare ordinul implicit este *INC*.

Comanda *CALL* este un *JMP* la care se adaugă înscrierea în stivă (*PUSH*) a adresei curente incrementate (figura 6.65). *CCALL* este un *CALL* condiţionat. Comanda *RET* presupune un *JMP* la adresa stocată în vîrful stivei (întoarcerea din subrutină) şi eliminarea (*POP*) din *LIFO* a înregistrării din vîrf (ultima). Pentru *RST* se activează intrarea *E'* a *MUX* care în schema anterioară (figura 6.62) era considerată cuplată la masă. □

**Exemplul 6.30** Să presupunem că avem de scris un microprogram care evaluează expresii de tipul

$$E = a_0 b_0 + a_1 b_1 + \ldots + a_n b_n.$$

Vom scrie foarte uşor acest microprogram dacă vom putea apela o rutină de înmulţire şi una de adunare folosite, evident, şi în alte microprograme. Nu trebuie uitat nici faptul că cel ce a scris rutina de înmulţire a apelat, tot pentru simplitatea programului, la o rutină de adunare. De asemenea, este bine să gîndim şi microprogramul pentru expresia *E*, tot ca pe o subrutină, pentru ca un alt utilizator să o poată apela într-un microprogram cu o funcţionalitate mai complexă.

Necesitatea unui mecanism care să permită realizarea unor subrutine, ce folosesc subrutine şi sînt folosite ca subrutine, este evidentă. Cititorul poate realiza ca pe un amplu exerciţiu subrutina pentru expresia *E* şi subrutinele pe care aceasta le presupune. □

Subrutina într-un microprogram nu impune decît salvarea adresei de retur în microprogramul principal. Acest lucru se datorează faptului că saltul la o subrutină se face exclusiv sub controlul microprogramului şi pot fi alese momente în care alte mărimi de stare nu sînt esenţiale. În cazul programelor rulate de un

microprocesor apariţia unei întreruperi putea surprinde programul în orice moment, fapt ce impunea şi salvarea registrului de stare (cel puţin).

Atunci cînd un procesor microprogramat implementează un set uzual de instrucţiuni, necesitatea unui *CROM* cu *LIFO* nu este întotdeauna stringentă. Numai cînd aplicaţia este nestandard şi implică microprograme complexe, o stivă internă îşi poate dovedi utilitatea.

• **Facilităţi aritmetice suplimentare** pe lîngă *RALU* sînt din ce în ce mai des introduse pentru creşterea performanţelor de viteză şi de microprogramabilitate. O subrutină poate fi înlocuită cu o structură fizică rezultînd creşteri spectaculoase în performanţe. Putem exemplifica prin *circuite de rotire sau deplasare* (barrel shifter), de *înmulţire*, sau alte funcţiuni, de regulă aritmetice, care por fi conectate "în paralel" cu *ALU* din *RALU*. Nu vom intra în astfel de detalii deoarece ţin mai mult de domeniul *arhitecturii sistemelor de calcul*.

### 6.7.3 Microprogramarea dinamică

**Definiţia 6.14** Un *procesor microprogramat dinamic* este caracterizat prin înlocuirea memoriei *ROM* de microprograme din *CROM* cu o memorie *RAM*. □

Care sînt motivaţiile ce impun o astfel de abordare a maşinilor microprogramate?

• **Compatibilitatea tehnologică** realizată în ultimele decenii *a permis* o astfel de abordare. Primele maşini microprogramate, construite în anii '50, foloseau o memorie *ROM* realizată pe toruri sau bare magnetice, deci într-o tehnologie net diferită de a structurilor digitale cu care era intim conectată (registre, circuite logice combinaţionale). Acest fapt limita dimensiunea şi performanţele *ROM*-ului. Evoluţia tehnologică a permis realizarea, în ultimele două decenii, a memoriei de microprograme într-o tehnologie identică cu a circuitelor ce funcţionau în acelaşi sistem. Drept consecinţă, performanţele de viteză şi complexitate au crescut, generînd noi probleme şi oferind noi posibilităţi.

• **Testarea microprogramelor** devine o problemă esenţială odată cu creşterea numărului şi dimensiunii lor. Folosirea din start a memoriilor *ROM*, chiar reprogramabile, nu este posibilă deoarece este iluzoriu să ne imaginăm că vom putea proiecta şi realiza dintr-un început fără eroare un set de microprograme. Pentru un singur bit eronat (la proiectare, la translatarea binară sau la programarea electrică propriuzisă) se impune ştergerea unui cip de memorie cu mii sau zeci de mii de biţi şi rescrierea (cu riscul unor alte erori). În consecinţă, pînă în momentul în care microprogramele funcţionează conform intenţiilor noastre, este necesar un suport care să permită scrierea şi modificarea foarte rapide şi cu un efort minimal. Memoria *RAM* poate oferi aceste facilităţi.

• **Evaluarea arhitecturii actualizate printr-un set de microprograme** este o problemă cel puţin la fel de critică. Punerea în funcţiune a fiecărui microprogram în parte, reprezintă numai o primă etapă în finalizarea unei maşini. Cel mai important lucru este ca arhitectura definită să atingă performanţele scontate în rezolvarea unor probleme concrete. Nu este întotdeauna sigur că setul de



instrucţiuni (funcţiuni) realizat este utilizabil performant în situaţii reale. Acest lucru este şi foarte greu de estimat teoretic. Numai măsurători concrete, cu programe standard pe structuri de date tipice, pot valida o anumită performanţă. Dacă ea nu este dovedită vor trebui redefinite măcar parţial unele instrucţiuni sau se vor impune altele noi, prin combinarea cărora să putem rezolva mai eficient probleme concrete. De multe ori o soluţie arhitecturală teoretică, gîndită foarte elegant, din punct de vedere formal, se dovedeşte ineficientă odată pusă în funcţie. Realitatea presupune soluţii ad-hoc, greu previzibile teoretic, dar deosebit de eficiente în cazuri critice.

Testarea diferitelor soluţii arhitecturale va cere de asemenea un suport flexibil pentru microprograme în faza de proiectare şi evaluare. Microprogramarea dinamică oferă acestă flexibilitate.

• **Accesul utilizatorului la nivelul microprogramării** este o altă cerinţă ce poate impune microprogramarea dinamică. Există utilizatori care pentru a creşte performanţe în zone funcţionale critice ale sistemelor pe care le proiectează sînt nevoiţi să-şi definească şi să realizeze funcţiuni microprogramate pe care fabricantul nu le-a prevăzut în setul de instrucţiuni. La nivelul programării, oricît de abil ar fi scrise programele, pe o maşină dată, nu se poate creşte performanţa în absenţa unor instrucţiuni specializate domeniului în cauză. (Spre exemplu, un microprogram care înmulţeşte matrici va fi întotdeauna mai rapid de *3* pînă la *5* ori, cel puţin, decît un program ce rulează pe acelaşi procesor cu setul de instrucţiuni oferit de fabricant pentru aplicaţii de uz general.)

Au fost construite, în consecinţă, procesoare la care, cel puţin o parte din memoria de microprograme este de tip *RAM*. Facilitatea de *WCS* (writeable control store) permite utilizatorului să completeze setul de instrucţiuni cu unele ce-i sînt necesare.

• **Necesitatea unei memorii de microprograme de mare capacitate** se impune ca o consecinţă a motivaţiilor anterioare. Este o tendinţă firească aceea ca funcţiuni realizate la nivelul programării să tindă, din considerente de performanţă, către un nivel mai coborît, care este, de altfel şi mai natural. Orice funcţie trebuie să tindă, spre a fi implementată, la nivelul structural cel mai coborît ce o admite.

Uneori dimensiunea ansamblului microprogramelor este atît de mare încît ele nu mai pot fi simultan stocate în memoria de microprograme. Actualizarea lor nu poate fi făcută dinamic decît dacă *ROM*-ul din *CROM* este înlocuit cu *RAM*.

• **Modificările structurale impuse de microprogramarea dinamică** sînt, în principal, două:

- apariţia mecanismelor ce permit scrierea în memoria *RAM* de microprograme;

- rezolvarea problemei alocării dinamice a spaţiului din *RAM* unde un microprogram este implantat.

În figura 6.66 sînt reprezentate soluţiile acestor două probleme. Prin *C* s-a notat structura din figura 6.65 cu excepţia *ROM*-ului şi transcodorului $TC_2$. Microprogramele sînt stocate în *RAM*-ul de *1 K* cuvinte de *64* biţi. Încărcarea fiecărei linii de microprogram se face prin patru scrieri, folosind căile *ADR*, *DATE* şi semnalul *WE'* cu *S = 1* (semnalul de selecţie al *MUX L*). *DMUX* distribuie, în

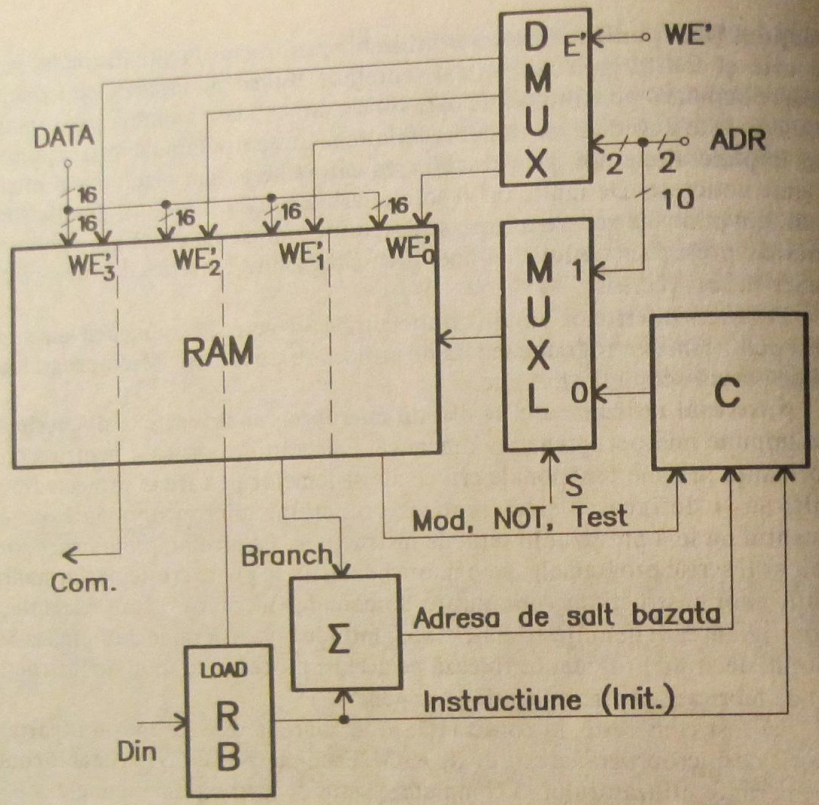



**Figura 6.66**

funcţie de cei mai puţin semnificativi doi biţi ai *ADR*, semnalele $WE'_0$, ..., $WE'_3$. Dacă $S = 0$ sistemul rulează microprogramele existente, anterior încărcate.

Cea de a doua problemă se pune datorită faptului că un microprogram poate fi implantat în memorie, la momente de timp diferite, în locuri diferite (începînd de la adrese diferite). Dificultatea apare din cauza adreselor de salt ce depind de zona din memorie în care microprogramul se află. Pentru a rezolva această problemă salturile se realizează *bazat*, în sensul că în *RB* (registrul de bază) se înscrie codul instrucţiunii (adresa de început a microprogramului) iar adresele de salt se obţin prin sumarea conţinutului acestui registru cu adresa de salt din microprogramul scris ca şi cum ar fi implantat la adresa zero. Adresele de salt în microprograme sînt de fapt salturi relative la adresa iniţială. Sumatorul $\Sigma$ permite calculul adresei de salt efective pornind de la *Branch* şi conţinutul *RB*.



### 6.7.4. Concluzii

Tehnica microprogramării a permis o creştere foarte puternică a performanţelor sistemelor de calcul începînd din anii '50. În anii '70 se părea că tehnologiile bazate pe microprocesoare vor face ca microprogramarea să devină un procedeu desuet. Nu s-a întîmplat aşa, din mai multe motive:

- unităţile centrale ale calculatoarelor puternice, în anii '70 şi '80, au continuat să fie microprogramate;

- această tehnică permite realizarea unor performanţe ridicate în sisteme orientate pe funcţiuni (ex: procesoare Fourier) sau orientate pe limbaj (ex: procesoare *LISP*);

- chiar evoluţia structurii interne a microprocesoarelor s-a realizat printr-o tendinţă evidentă către mecanismele sistemelor microprogramate;

- în deceniul '90 unităţile de procesare vor tinde către "microprocesoare" cu structură internă de procesor microprogramat, în tendinţa de a anula diferenţa dintre procesoarele folosite în sisteme cu performanţe diferite.

Microprogramarea este rezultatul unei tendinţe fireşti de migrare funcţională către nivele structurale cît mai coborîte. Pe măsură ce anumite funcţiuni se limpezesc şi-şi dovedesc utilitatea apare tendinţa firească de a fi realizate cît mai performant şi, drept consecinţă, la un nivel structural cît mai profund. De regulă, programele tind să se întrupeze în structuri funcţionale atunci cînd şi-au dovedit utilitatea.

Această tehnică a fost multă vreme o tehnică "de firmă", la care utilizatorul nu avea acces. A rămas în mare măsură aşa, dar există tendinţa de fi pusă şi la dispoziţia celor ce-şi vor putea dezvolta aplicaţii ce impun o bună cunoaştere a maşinii pentru obţinerea performanţei. Cu maşinile microprogramate ne aflăm la limita dintre structurile fizice şi cele informaţionale. Microprogramul este o structură informaţională încă puternic legată de una fizică. De aici dificultatea de a folosi această tehnică, dar şi performanţa la care uneori ajungem.

## 6.8 Concluzii

Clasa procesoarelor încheie un proces de structurare declanşat conform principiului expus în capitolul 2 al acestei lucrări. Încheie şirul sistemelor digitale la care *funcţia este dictată numai de structura fizică*. Deschide, totodată, seria *sistemelor în care funcţia este condiţionată şi de structuri simbolice* de tipul microprogramelor sau al programelor pe care le putem pune sub un nume generic, acela de *informaţie* (pentru detalii vezi [Drăgănescu '91], [Ştefan '91]).

La nivelul sistemelor de ordinul trei informaţia începe să ia controlul funcţiei sistemului. Structura fizică oferă o *clasă potenţială de funcţiuni* care *se*

*actualizează* prin intermediul unor structuri simbolice, ce pot acţiona - informaţia. Va deveni din ce în ce mai puţin important, în continuare, mecanismul de structurare fizică în ordine prin bucle de reacţie, deoarece va spune din ce în ce mai puţin despre funcţionalitatea sistemului.

Inginerul electronist care construieşte hardware-ul unei maşini operează cu *clase* de arhitecturi numai, urmînd ca prin microprogramare, sau, la sisteme de ordinul patru, prin programare, să fie actualizată numai *o anumită arhitectură*.

Am prezentat procesoarele ca rezultînd dintr-o creştere organică, în virtutea unui principiu de structurare. Poate aşa au apărut în primele etape. Astăzi structura lor este dictată din perspectiva unor funcţiuni sau clase de funcţiuni pe care trebuie să le îndeplinească. De multe ori funcţia de procesare este îndeplinită de structuri ce au un ordin superior lui trei (chiar în tratarea noastră, *CROM*-ul cu stivă creşte ordinul procesorului, din care face parte, cu o unitate). În consecinţă, aprofundarea studiului procesoarelor poate fi continuată numai în cadrul ştiinţei calculatoarelor. Conceptul de procesor poate aparţine unui curs de circuite, dar dezvoltarea lui, cu studierea celor mai noi tendinţe, depăşeşte cadrul unui astfel de curs.

Am studiat în acest capitol două structuri limită: microprocesorul şi procesorul microprogramat. Reamintim faptul că circuitele reale împrumută caracteristici ale ambelor modele teoretice. Dar trebuie să subliniem şi tendinţa, ca în lumea procesoarelor, să se tindă către un dispozitiv unic, utilizabil în orice tip de maşină, indiferent de complexitatea acesteia. Termenul de microprocesor îşi va pierde sensul pentru că nu va mai distinge un sistem dintr-o clasă mai mare. Vom folosi, sper, curent termenul de *procesor* şi ne vom referi, în curînd, la un dispozitiv standard.

Procesoarele ating nivelul unei maxime flexibilităţi funcţionale. *Logica programată* este numai un exemplu în acest sens. Dar vom menţiona, în acelaşi timp, şi faptul că performanţa este alterată de o serie de acţiuni şi mecanisme parazite. Să ne reamintim numai, programul scris în paragraful 6.6.5 sau microprogramul asociat aceleeaşi funcţii în *Exemplul 6.28*. În programul amintit numai *o instrucţiune din cele şapte* efectua adunarea ce reprezenta scopul programului; restul instrucţiunilor manipulau datele astfel încît numerele de adunat "să ajungă" la intrarea circuitului de sumare din microprocesor. De asemenea, în microprogramul citat, numai o microinstrucţiune din cele 9 (exceptînd rutina *FETCH*) realiza adunarea celor două numere. Acesta este preţul plătit posibilităţii de a efectua *o clasă foarte largă de funcţiuni folosind o structură fizică unică* în calitate de *maşină universală*.



# 7

# CONCLUZII

Oprim în acest punct studiul evoluţiei structurale a sistemelor digitale, după ce am ajuns la sistemele de ordinul trei. Cele de ordin superior lui trei sînt dominate funcţional de informaţie, în variantele ce sînt utilizate în contextul tehnologic actual. Elementul tipic al sistemelor de ordinul patru, calculatorul, depăşeşte cadrul acestui curs ce studiază, cu preponderenţă, *structurile fizice ale sistemelor digitale*. Viziunea *funcţională* şi, în consecinţă, cea *arhitecturală* domină gîndirea sistemelor de ordin superior. În acest cadru abordarea structurală trece pe planul doi şi capătă două aspecte: unul legat de structura fizică (hardware) iar altul legat de structura informaţională (software). Aspectele funcţionale au fost secunde în expunerea anterioară; ele devin semnificative numai, acum, în finalul expunerii.

- **Corelaţia între nivelul structural** (ordinul sistemului) *şi o anumită clasă de funcţiuni* ne apare evidentă:

  - sisteme de ordinul zero - circuite logice combinaţionale,
  - sisteme de ordinul unu - circuite de memorare,
  - sisteme de ordinul doi - automate finite elementare,
  - sisteme de ordinul trei - procesoare.

Un sistem de un anumit ordin acceptă implementarea funcţiilor asociate şi sistemelor de ordin inferior lui. Vom putea adăuga acestor corespondenţe, fără a fi necesare foarte multe explicaţii, şi pe următoarea:

- sisteme de ordinul patru - calculatoare,

reprezentînd în figura 7.1 cele patru tipuri teoretic posibile, dintre care varianta *b* este cea uzuală. (*P + ROM* reprezintă un calculator cu programele scrise în *ROM*;

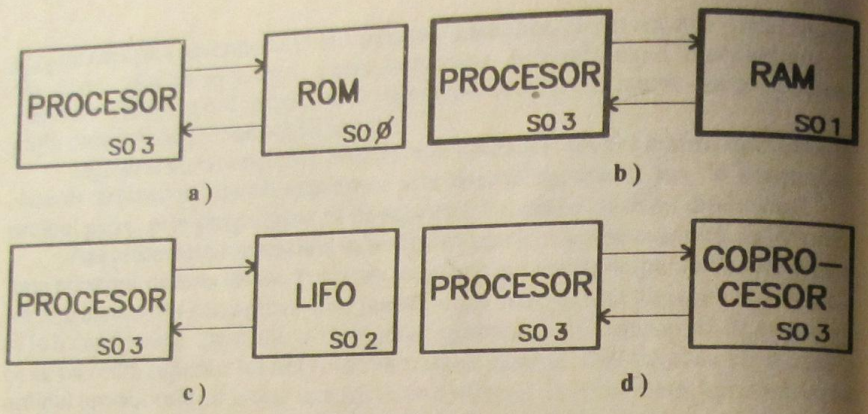



**Figura 7.1**

*P + LIFO* reprezintă un sistem simplu de evaluat expresii; *P + COPROCESOR* reprezintă un sistem în care anumite funcţiuni critice sînt transmise către un coprocesor, spre exemplu unul aritmetic.)

Corelaţia evidenţiată este esenţială pentru că stabileşte nivelul structural minim pe care o anumită funcţie poate fi implementată. De regulă, performanţa maximă de viteză se obţine pe acest nivel minimal. Flexibilitatea poate fi o capcană ce ne atrage către nivele de ordin mare. Logica programată, ca tehnică ce presupune *SO4*, poate aborda oricare din funcţiunile sistemelor de ordin inferior, cu o flexibilitate maximă, dar, din păcate, cu performanţe de viteză minimale. *Structura fizică universală* actualizabilă prin programe (informaţie) se poate mula pe orice funcţie dacă timpul nu este un parametru important.

La cealaltă extremă, şi un circuit combinaţional se poate institui, sub forma unui *ROM*, într-o structură universală (figura 7.2), dar cu preţul unei *complexităţi* inacceptabil de mari (conform paragrafului 3.6.1). Această structură oferă performanţa de viteză maximă, dar o complexitate ce depinde exponenţial de cele $n + m$ intrări. Toată dezvoltarea realizată în acest curs a fost necesară pentru că nu

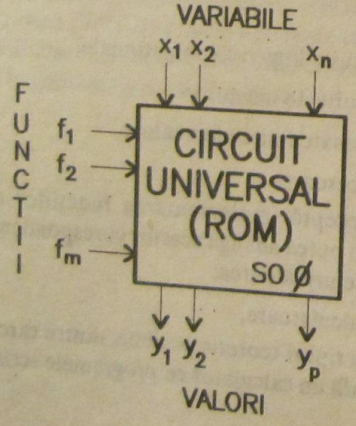



**Figura 7.2**



cunoaştem o soluţie digitală, acceptabilă din punctul de vedere al complexităţii, sub forma unui *CLC* pentru funcţiunile cerute sistemelor digitale.

● **Trecerea de la o structurare echilibrată la una neechilibrată** în procesul de creştere al ordinului sistemelor digitale este, de asemenea, un fenomen ce trebuie pus în evidenţă. Facem acest lucru prin reprezentarea unor structuri tipice de ordinul *1, 2, 3* şi *4* în figura 7.3. Remarcăm echilibrul şi simetria latch-ului, ca reprezentant tipic pentru *SO1*. Mai apoi asimetria automatului finit (*SO2*), deoarece pe intrarea latch-ului slave nu există un *CLC* explicit; pe buclă se transferă încă semnale de acelaşi fel. Procesorul este deja complet dezechilibrat, fiind format din două structuri net distincte, una rezultată sub presiunea optimizării din perspectiva spaţiului stărilor (*RALU*), iar cealaltă rezultînd ca o soluţie eficientă pentru un automat cu intrarea foarte complexă (*CROM*). Nici semnalele care se propagă între subsisteme nu mai sînt de acelaşi tip, comenzi într-un sens şi indicatori în celălalt sens. În ultimul caz în bucla procesorului se află un sistem cu două ordine inferior - memoria.

Accentuarea caracterului neechilibrat al structurii este parţial responsabilă de apariţia şi creşterea rolului informaţiei în structurile digitale, cu toate avantajele imediate şi dezavantajele din perspectivă.

Există şanse, şi chiar o uşoară tendinţă (vom vedea în capitolul următor), de restabilire a unui proces de structurare echilibrată la nivelul sistemelor de ordinul doi, menţinînd aceste structuri în atenţia noastră pentru o perspectivă mai îndepărtată.

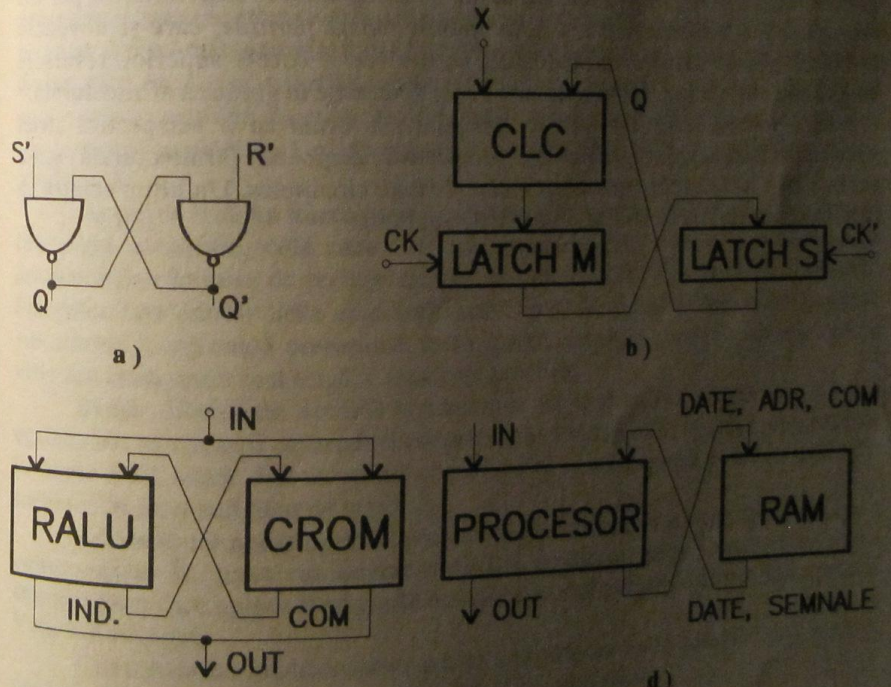



**Figura 7.3**

Dacă vom putea extinde structurarea echilibrată şi peste nivelul *SO2*, atunci vom cuceri spaţii utile pentru arhitecturile digitale de mare performanţă. În acest moment însă, nu putem să nu remarcăm faptul că, nu numai structural, *RALU* şi *CROM* diferă, dar şi funcţional distincţia este imensă, deoarece prima este un *automat efector* (de prelucrare) iar cea de a doua structură are funcţiuni de *control*. Iar în cazul *SO4* discrepanţa creşte în mod alarmant, deoarece în procesor la fiecare ciclu, aproape întreaga structură este solicitată iar în *RAM*, în fiecare ciclu, nu este activată decît cel mult o locaţie de memorie. Dacă, *RAM*-ul are capacitatea de *1 M* cuvinte, atunci, maxima solicitare se face într-o proporţie egală cu $10^{-6}$. Putem spune deci, cu eroare de $10^{-6}$, că *RAM*-ul este o structură în care nu se întîmplă nimic. Dacă în procesor un automat *face* iar celălalt numai *controlează*, într-un calculator, procesorul *lucrează* iar memoria *nu face nimic*, cu o aproximaţie cu atît mai bună cu cît memoria este mai mare.

Revenirea către procese de structurare echilibrată devine un imperativ al gîndirii structurale şi arhitecturale în domeniul sistemelor digitale.

Mecanismele de structurare neechilibrată introduc în sistemele digitale structuri care se comportă din ce în ce mai parazit. Latch-ul slave permite numai închiderea corectă a buclei, din punctul de vedere al propagării semnalelor; nu are şi un rol funcţional. *CROM*-ul asigură o înlănţuire ce activează coerent conţinutul amorf al *RAM*-ului din *RALU*. Despre *RAM*-ul din *SO4* am vorbit anterior. Efortul de gîndire pe care va trebui să-l facem trebuie direcţionat astfel încît, orice creştere a complexităţii structurale să fie motivată strict funcţional, nu să compenseze efecte parazite cum o face în mare măsură acum. Neomogenitatea structurilor a permis segregarea, în anumite zone, a prea multor *funcţii parazite*, care-şi dovedesc utilitatea numai datorită incapacităţii noastre de a corela superior resursele structurale puse în joc; poate şi datorită unei deficienţe în gîndirea arhitecturală.

Nu trebuie să uităm însă că am pornit la drum cu o perspectivă strict structurală. Am neglijat gîndirea din perspectivă *funcţională* (arhitecturală) şi nu am înţeles pînă la capăt consecinţele *complexităţii* circuitelor. Un ultim capitol va încerca redirecţionarea modului nostru de abordare.



# 8

# REŢELE FUNCŢIONALE COMPLEXE

Inginerii electronişti sînt încă obsedaţi de ideea că trebuie concepute sisteme, procedee şi tehnologii care să poată rezolva un spectru foarte larg de funcţiuni, dacă nu chiar orice funcţie formal definibilă. Progresul tehnologic şi teoretic este marcat de tendinţa de a oferi soluţii cu caracter universal. În ceea ce am expus pînă acum *nu am impus restricţii funcţionale*, am gîndit structuri bune pentru orice aplicaţie. Nu se poate urmări însă performanţa pe un astfel de front, cu o lărgime nerestricţionată.

Ar putea fi de un imens interes preocupările pentru restringerea la seturi de funcţiuni esenţiale, cele care într-adevăr se dovedesc utile în structurarea arhitecturilor impuse de cerinţe umane, naturale sau sociale. Nu cumva spectrul funcţiilor într-adevăr utile este mult mai îngust decît cel al funcţiilor posibile. Abordarea funcţională presupune, într-o primă etapă, segregarea acestor funcţii utile din clasa, mult mai largă, a funcţiilor posibile.

Şansa oferită de această concentrare va trebui fructificată în vederea optimizării în condiţii de maximă complexitate. Dacă *m*, din sistemul reprezentat în figura 6.68, scade drastic atunci s-ar putea să existe o şansă pentru a optimiza sistemul cu un *n* suficient de mare.

Ultimele tehnologii permit realizarea unor structuri cu zeci de milioane de componente. În acest caz devine esenţială studierea unor structuri de marc complexitate care nu pot fi stăpînite numai cu procedee simple de genul diagramei Veitch-Karnaugh.

Concentrarea funcţională va permite abordarea unor complexităţi mari în contextul în care vom scăpa de obsesia maşinilor universale.

În acest capitol vom analiza complexitatea reţelelor orientate pe funcţiuni pentru a pune în evidenţă o problematică specifică, marcînd diferenţa faţă de preocuparea de a reduce, spre exemplu, *2* porţi din *5*, prin procedee aplicabile *oricărei* funcţii logice. Ne va interesa cum să reducem complexitatea ($S(n)$) şi adîncimea ($D(n)$) unei structuri la care complexitatea intrării poate fi oricît de mare. Contează mai puţin optimizarea unui sumator complet de *2* biţi şi mult mai mult cum depinde complexitatea unui sumator de $n$ biţi de valoarea lui $n$. Şi aceasta, numai dacă funcţia de sumare este una din cele de care sîntem, sau merită să fim, interesaţi.

## 8.1 Ce este un sistem complex?

Să ne reamintim notaţia introdusă în *Anexa 9.1*, unde $f(n) = O(g(n))$ pentru $n \geq n_0$ dacă $f(n) < c\,g(n)$, $c$ fiind o constantă. În tot ce am studiat pînă in acest moment am fost preocupaţi să minimizăm efecte introduse de constanta $c$. *Pentru sistemele complexe $c$ este nesemnificativ*, esenţială este dependenţa funcţională $g(n)$.

**Exemplul 8.1** Să presupunem trei tipuri de dependenţe: $f_1(n) = O(\log n)$, $f_2(n) = O(n)$şi $f_3(n) = O(2^n)$. O anumită combinaţie de constante $c_1$, $c_2$ şi $c_3$ ne poate oferi reprezentarea din figura 8.1. Pentru $n < n_1$ valoarea lui $f_3(n)$ este cea mai mică, iar numai pentru $n > n_2$ valoarea lui $f_1(n)$ reprezintă complexitatea minimă. Dacă aceste dependenţe se referă la complexitatea cu care realizăm o anumită funcţie digitală, atunci pentru valori suficient de reduse ale lui $n$ poate fi optim chiar un algoritm exponenţial. □

În mod evident, ceea ce gîndim la complexităţi mici nu este relevant pentru complexităţi mari, din cauza acestor dependenţe care în funcţie de constantele asociate pot da imagini complet diferite pentru diferite valori ale lui $n$. Procedee de implementare utilizabile pentru un $n$ limitat se pot dovedi falimentare dacă trecem de o anumită mărime a lui $n$. Structuri şi algoritmi acceptabili pentru $n < n_0$, devin

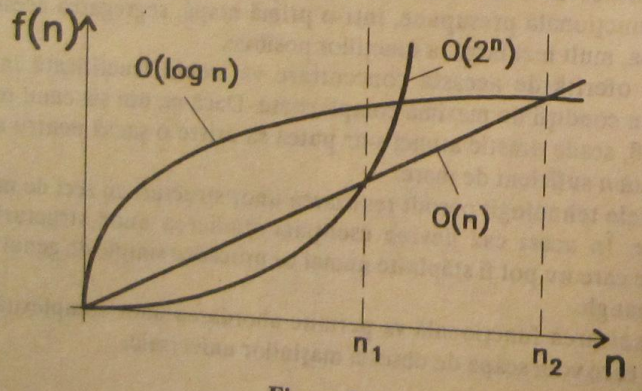



**Figura 8.1**



inabordabile din punct de vedere tehnologic pentru $n > n_0$. Rezultă că modul nostru de gîndire are un punct de comutare în funcţie de un $n_0$ asociat, de fiecare dată cu altă valoare, diferitelor funcţiuni realizate în tehnologii digitale.

● **Ponderea pe care o are un subansamblu** într-un sistem complex depinde şi ea foarte mult de $n$, în sistemele complexe ($n > n_0$) impunîndu-se strategii diferite de cele utilizate la nivelele de complexitate redusă ($n < n_0$).

**Exemplul 8.2** Pornind de la numărătorul sincron de $n$ biţi reprezentat în figura 5.19 şi considerînd complexitatea unei porţi egală cu numărul de intrări pe care le are, rezultă complexitatea numărătorului de forma

$$S_N(n) = 0{,}5\,n^2 + 25{,}5\,n - 1 = O(n^2).$$

Se pune întrebarea: de cine este dată complexitatea circuitului? De reţeaua de porţi sau de reţeaua de bistabili de tip $T$? Răspunsul este dependent de $n$. În jurul valorii de $n_0 \approx 50$ răspunsul comută. Pentru $n < n_0$ ponderea esenţială în complexitatea numărătorului o au bistabilii, iar pentru $n > n_0$ complexitatea este determinată de reţeaua de porţi. Deci dependenţa pătratică ne va incomoda numai la complexităţi mari. Vom regîndi obligatoriu reţeaua de porţi la $n > n_0$; o puteam face şi înainte dar fără a obţine efecte spectaculoase. □

● **Accepţiunea dată termenului de complexitate** va putea fi, credem, mai uşor înţeleasă din perspectiva exemplelor anterioare.

**Definiţia 8.1** Un sistem este complex atunci cînd optimizarea lui se poate reduce la optimizarea funcţiei $g(n)$ din $S(n) = O(g(n))$, unde $S(n)$ reprezintă complexitatea sistemului în funcţie de dimensiunea $n$ a intrării. □

Progresul tehnologic permite ca tot mai multe funcţii digitale să poată fi realizate pentru $n > n_0$, intrînd astfel în *clasa funcţiilor complex realizabile*. Dacă toate acestea sînt funcţiuni utile, este o altă problemă ce se află sub incidenţa *electronicii funcţionale*.

Nu avem încă o teorie generală a funcţiilor digitale complexe. Vom expune subiectul folosind cîteva exemple semnificative ce vor fi reluate în mai multe maniere de implementare.

## 8.2 Reţele combinaţionale

Studiul reţelelor combinaţionale, în cazul general, oferă pentru o adîncime constantă ($D(n) = O(1)$) o complexitate exponenţială ($S(n) = O(2^n)$). Există însă funcţiuni care se comportă mai puţin violent, permiţînd soluţii tehnologic realizabile, ce pot suporta chiar optimizări, dacă renunţăm la adîncimea constantă în favoarea unor adîncimi acceptabile (spre exemplu $D(n) = O(P(\log n))$, unde $P(\log n)$ este un polinom de logaritmi). Nu ştim cît de extinsă este clasa acestor funcţiuni; o vom exemplifica cu *reţele de calcul al prefixelor*, cu *reţele de sortare* şi cu *reţele de permutare*.

### 8.2.1 Reţele de calcul al prefixelor

Vom defini o *clasă de funcţiuni*, care va putea fi implementată cu acelaşi algoritm de structurare indiferent de funcţia concretă impusă de aplicaţie.

**Definiţia 8.2** Dîndu-se variabilele $x_1, x_2, ..., x_n$, ce pot lua valori în aceeaşi mulţime de configuraţii binare, se definesc funcţiile de tip *prefix*, $f_1, f_2, ..., f_n$, după cum urmează:

$$f_1(x_1) = x_1$$
$$f_i(x_1, x_2, ..., x_i) = f_2(f_{i-1}(x_1, x_2, ..., x_{i-1}), x_i),$$

pentru $i = 2, 3, ..., n$. □

**Exemplul 8.3** O particularizare posibilă a definiţiei anterioare poate fi făcută pentru $x_i \in \{0, 1\}$ şi $f_2(x_1, x_2) = AND(x_1, x_2) = x_1 x_2$. Rezultă circuitul combinaţional asociat numărătorului sincron, reprezentat în figura 5.19. □

Schema bloc pentru o *reţea de calcul al prefixelor* cu $n$ intrări ($RCP_n$) este reprezentată în figura 8.2, unde notaţiile au semnificaţia dată de *Definiţia 8.2*. Care ar putea fi structura internă a acestui circuit? Dar complexitatea sau adîncimea?

O variantă posibilă este reprezentată în figura 8.3.

**Propoziţia 8.1** În cel mai favorabil caz complexitatea circuitului din figura 8.3 este $S_{P1}(n) = O(n^2)$. □

*Demonstraţie.* În cel mai favorabil caz, complexitatea circuitului $f_i$ (figura 8.3) este $S_f(i) = O(i)$, pentru $i = 2, 3, ..., n$ (ca în situaţia din *Exemplul 8.3*). Rezultă:

$$S_{P1} = c(2 + 3 + ... + n) = O(n^2). \square$$

Adîncimea circuitului poate fi, tot în cel mai favorabil caz, constantă. Este cazul *Exemplului 8.3*.

Modul particular în care a fost definită mulţimea funcţiilor prefix permite şi o altă implementare, sub forma celei din figura 8.4, unde s-a folosit, ca modul, circuitul asociat funcţiei $f_2$. În acest caz este evident că $S_{P2}(n) = O(n)$ iar $D_{P2}(n) = O(n)$. Scăderea complexităţii a fost plătită prin creşterea adîncimii. Există însă o teoremă (cea a lui Spira) care ne asigură că vom putea găsi şi o soluţie

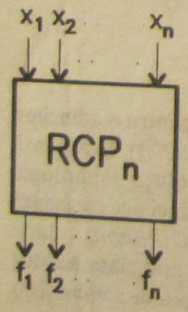



Figura 8.2

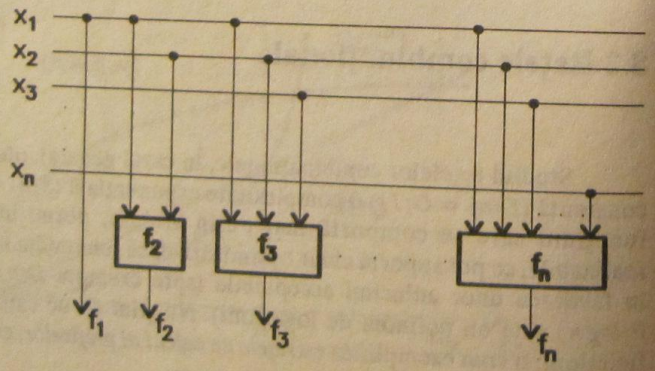



Figura 8.3



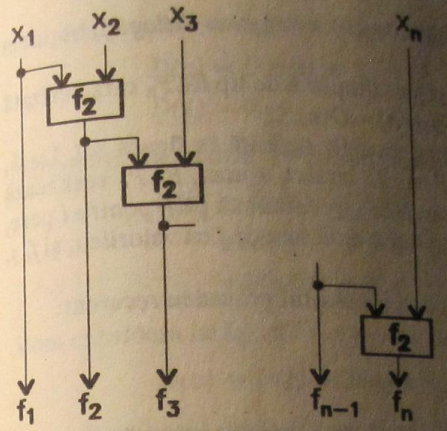



Figura 8.4

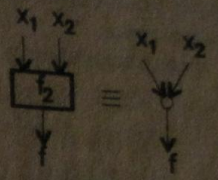



Figura 8.5

pentru care $D(n)$ să fie logaritmică. Problema ce se pune este dacă nu va creşte inceptabil de mult complexitatea.

Facem, în continuare, convenţiile din figura 8.5, pentru a utiliza notaţii şi reprezentări mai simple. Considerăm că $f_2(x_1, x_2) = f(x_1, x_2)$ este *funcţia caracteristică* a reţelei, avînd asociat modulul elementar folosit la structurarea sistemului.

*Atenţie!* În continuare vom considera că $n$ este de forma $2^m$, unde $m$ este un întreg. Demonstraţiile sînt mai uşor de făcut, fără a se altera esenţa rezultatelor obţinute. Atunci cînd $n$ nu va fi considerat o putere a lui doi, acest lucru va fi specificat.

**Exemplul 8.4** O reţea $RCP_8$ de adîncime logaritmică este reprezentată în figura 8.6. Complexitatea acestei reţele, în cazul general, este

$$S_{P3}(n) = n \log_2 n - (n - 1) = O(n \log n).$$

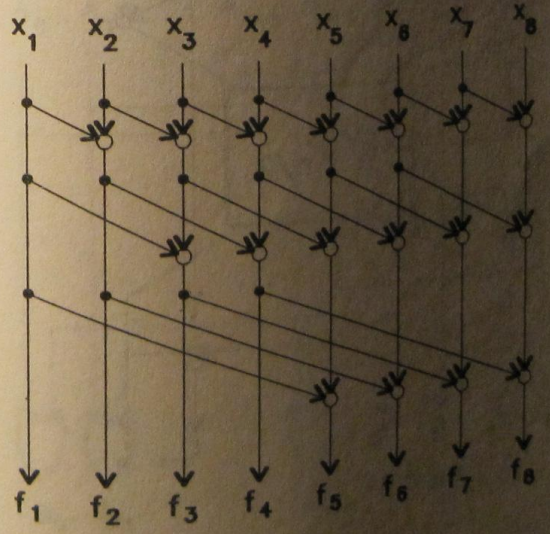



Figura 8.6

Rezultatul este încurajator, deoarece pentru $D_{P3}(n) = log_2 n = O(log n)$, obţinem o complexitate mai mică decît $S_{P1}(n)$. □

**Propoziţia 8.2** Se poate construi o reţea complexă de tip $RCP_n$ caracterizată prin $D(n) = D_{P4}(n) = O(log n)$ şi $S(n) = S_{P4}(n) = O(n)$. □

*Demonstraţie.* Pornim de la structura recursiv definită în figura 8.7. Dacă, $RCP_{n/2}$ este o reţea ce calculează prefixe cu $n/2$ intrări, atunci, $RCP_n$ realizează aceeaşi funcţie pentru $n$ intrări. Într-adevăr, $RCP_{n/2}$ realizează pe $f_i$ pentru $i$ pare, iar pentru valorile impare ale lui $i$ se mai aplică o dată funcţia $f$ cu valorile $x_i$ şi $f_{i-1}$, unde $i$ este impar.

Pentru calculul complexităţii vom folosi următorul procedeu recurent:

$$S(n) = S(n/2) + (n - 1),$$
$$S(1) = 0,$$

pornind de la analiza figurii ce reprezintă reţeaua. Prin inducţie rezultă:

$$S(n) = S(n/2) + (n - 1) = S(n/4) + (n/2 - 1) + (n - 1) =$$
$$= S(n/2^i) + n/2^{i-1} + n/2^{i-2} + \ldots + n/2^{i-i} - i.$$

Deoarece $n = 2^i$, pentru că vom aplica procedeul pînă cînd

$$S(n/2^i) = S(1)$$

rezultă:

$$S(n) = S(1) + n \sum_{k=0}^{-1+log_2 n} \left[\frac{1}{2^k}\right] - log_2 n = 2n - 2 - log_2 n = O(n).$$

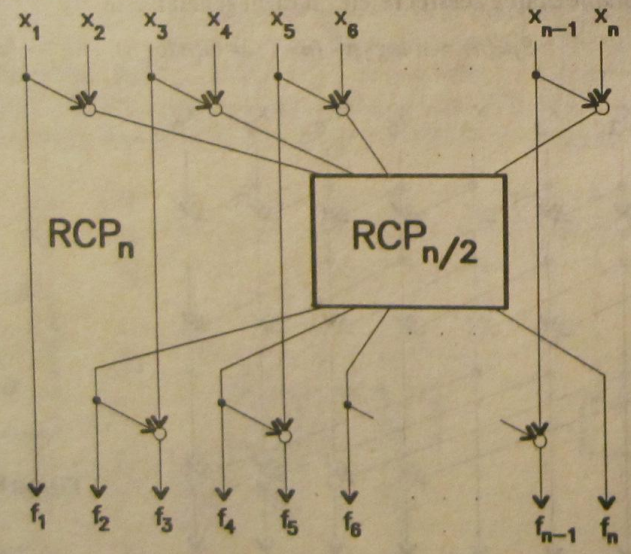



Figura 8.7



Adîncimea circuitului se calculează pornind de la relaţiile:

$$D(n) = D(n/2) + 2,$$
$$D(1) = 0,$$

extrase tot din analiza reprezentării reţelei. Procedînd similar scriem:

$$D(n) = D(n/2) + 2 =$$
$$= D(n/4) + 2 + 2 =$$
$$= D(n/2^i) + 2i.$$

Vom considera iarăşi că $i = log_2 n$ şi, în consecinţă:

$$D(n) = D(1) + 2log_2 n = 2log_2 n = O(log n).$$ □

Rezultatul obţinut este foarte important pentru că se referă la o clasă de reţele. În această a patra variantă dispunem de o reţea cu $S_{P4}(n) = O(n)$ şi $D_{P4}(n) = O(log n)$. Din punct de vedere al complexităţii nici nu ar mai avea sens să căutăm soluţii mai bune. Este puţin probabil ca pentru $n$ intrări să putem rezolva problema cu mai puţin de $O(n)$ circuite, iar o adîncime logaritmică, vom vedea, că reprezintă în actualele tehnologii electronice o limită sub care nu are rost să coborîm. La constante ne mai putem gîndi, dar nu formează obiectul acestui capitol.

**Exemplul 8.5** Folosind reţeaua perticularizată prin *Exemplul 8.3* se poate construi un circuit de incrementare a numărului binar de $n$ biţi, $A = A_{n-1} A_{n-2} \ldots A_1 A_0$. În figura 8.8 este reprezentat acest circuit. Regula după care a fost gîndit ţine cont de faptul că $A_i$, pentru $i = 1, 2, \ldots, n-1$, se complementează dacă există comanda $INC = 1$ şi toţi biţii $A_j$ cu $j<i$ sînt egali cu 1. Reţeaua $RCP_n$ *(AND)* este o reţea de *AND*-uri care pe ieşirile $f_i$ calculează această condiţie. Bitul $A_0$ comută la fiecare comandă de increment; prin $f_1(x_1) = INC$ se asigură acest

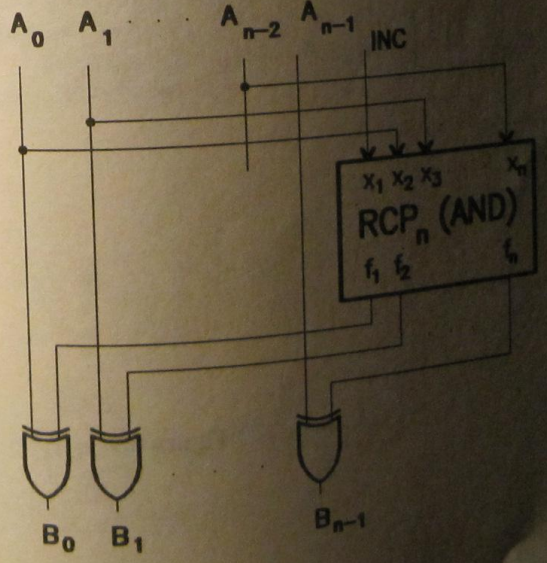



Figura 8.8

lucru. Pe lîngă reţeaua de *AND*-uri au mai fost adăugate numai $n$ circuite *XOR*, deci, complexitatea circuitului de incrementare va fi

$$S_{INC}(n) = O(n)$$

pentru o adîncime $D_{INC}(n) = O(\log n)$. Dacă acest circuit combinaţional este conectat în bucla unui registru de $n$ biţi, va rezulta un numărător sincron. Observăm că fiecare bistabil de tip $D$ va avea asociat un circuit *XOR*, ce-l transformă într-un bistabil de tip $T$. □

**Exemplul 8.6** O altă particularizare a $RCP_n$ o constituie cea care presupune că $x_i$ aparţine *reprezentărilor binare de m biţi*, iar $f_2(x_1, x_2)$ reprezintă *funcţia de sumare*. În acest caz

$$f_1(x_1) = x_1,$$
$$f_2(x_1, x_2) = x_1 + x_2,$$
$$f_3(x_1, x_2, x_3) = x_1 + x_2 + x_3,$$
$$\vdots$$
$$f_n(x_1, ..., x_n) = x_1 + x_2 + ... + x_n,$$

unde prin + am notat suma binară. □

Se pot imagina şi alte particularizări ale acestei reţele. Generalitatea acestui rezultat este foarte tentantă!

În figura 8.9 este reprezentată o reţea de circuite ce asigură propagarea depăşirii într-un circuit ce sumează numerele $A_{n-1} ... A_1 A_0$ şi $B_{n-1} ... B_1 B_0$. Prin $C_i$ este notat semnalul de carry generat de sumarea biţilor $A_{i-1}$ şi $B_{i-1}$, $C_0$ este un

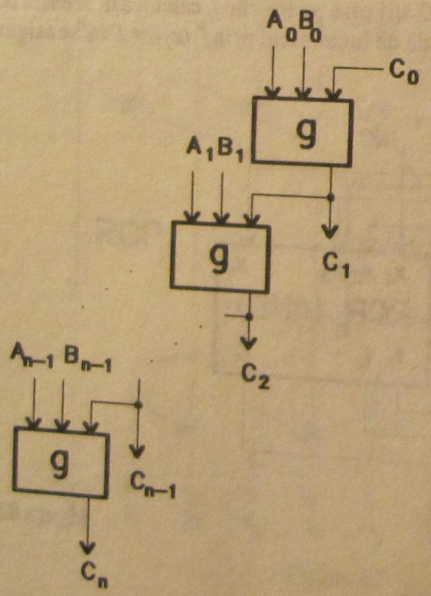



**Figura 8.9**



semnal ce se poate propaga de la o sumă anterioară, iar $C_n$ este depăşirea dată de sumator în ansamblul lui. Deci funcţia $g$ este de forma $g = A_iB_i + A_iC_i + B_iC_i = C_{i+1}$. Cititorul va găsi singur răspunsul la întrebarea: în ce măsură cele expuse în acest paragraf sînt utilizabile pentru a transforma această reţea într-una de adîncime logaritmică şi complexitate $O(n)$? Aceeaşi întrebare poate fi pusă referitor la circuitul din figura 5.107.

## 8.2.2 Reţele de sortare

Funcţia de sortare se referă, de regulă, la un şir de numere care trebuie ordonat crescător sau descrescător. Este o funcţie tipică într-un sistem de calcul.

**Definiţia 8.3** O reţea de sortare este un *CLC* care primind la intrare şirul de numere $<x_1, x_2, ..., x_n>$, generează la ieşire şirul $<y_1, y_2, ..., y_n>$ care reprezintă o permutare a şirului de intrare, astfel încît, pentru orice $i$, $i = 1, 2, ..., n$, $y_i \geq y_j$ dacă $i > j$, sau $y_i \leq y_j$ dacă $i > j$. □

Vom structura o reţea complexă folosind ca modul *sortatorul elementar* cu două intrări, $x_1$, $x_2$ şi două ieşiri $y_1$ şi $y_2$, caracterizate prin

$$y_1 = min(x_1, x_2),$$
$$y_2 = max(x_1, x_2);$$

sau

$$y_1 = max(x_1, x_2),$$
$$y_2 = min(x_1, x_2).$$

În cele ce urmează vom folosi prima variantă (sortarea în ordine crescătoare), pe care o vom simboliza ca în figura 8.10. Cu acest modul, un sortator pentru $n=4$, este reprezentat în figura 8.11. El are $S(4)=5$ şi $D(4)=3$. Care este regula după care putem construi un sortator pentru un $n$ oarecare?

**Definiţia 8.4** O reţea de sortatoare cu $n$ intrări, $x_i$ şi $n$ ieşiri, $y_i$ este o reţea de *sortatoare elementare* cu funcţiile de transfer de forma

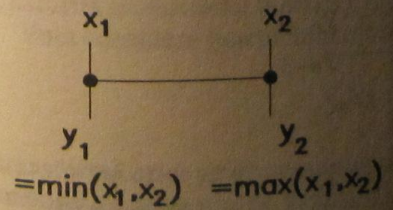



**Figura 8.10**

**Figura 8.11**

$$y_i = min(a, b) \mid max(a, b),$$

unde $a$ şi $b$ sînt definite prin

$$a = min(a, b) \mid max(a, b) \mid x_j$$
$$b = min(a, b) \mid max(a, b) \mid x_k$$

pentru $i = 1, 2, ..., n$ şi $j, k \in \{1, 2, ..., n\}$. □

**Exemplul 8.7** Pentru reţeaua din figura 8.11 vom scrie

$$y_1 = min(min(x_1, x_2), min(x_3, x_4))$$
$$y_2 = min(min(max(x_1, x_2), max(x_3, x_4)), max(min(x_1, x_2), min(x_3, x_4)))$$
$$y_3 = max(min(max(x_1, x_2), max(x_3, x_4)), max(min(x_1, x_2), min(x_3, x_4)))$$
$$y_4 = max(max(x_1, x_2), max(x_3, x_4)).$$ □

*Atenţie*! O reţea de sortatoare nu realizează întotdeauna funcţia de sortare.

**Propoziţia 8.3** O reţea de sortatoare generează la ieşirile $y_1, ..., y_n$ o permutare a intrărilor $x_1, ..., x_n$. □

*Demonstraţie*

**Definiţia 8.5** Un *nivel* de comparatoare este reprezentat de comparatoarele care pot efectua compararea în paralel asupra unui set de mărimi ce provine de la nivelul anterior. Intrarea în reţeaua de comparatoare reprezintă nivelul zero. □

La ieşirile *fiecărui* nivel de comparare, ce este caracterizat numai de funcţii de tipul *min* sau *max* cu un singur nivel, se regăsesc, într-o altă ordine, toate valorile de pe intrare. Deci după oricîte nivele vom regăsi toate valorile de intrare permutate. □

O regulă simplă de a construi o reţea de *sortatoare* care să se comporte ca o *reţea de sortare* este dată în figura 8.12, unde se arată cum se poate construi, recursiv, un sortator de $n$ numere ($RS_n$), folosind unul de $n-1$ numere ($RS_{n-1}$) şi $n-1$ sortatoare elementare. (Pentru $n=8$ este reprezentat acest tip de sortator în figura 8.13). Se calculează imediat că

$$S_{RS}(n) = S_{RS}(n-1) + (n-1) = O(n^2),$$
$$D_{RS}(n) = D_{RS}(n-1) + 2,$$

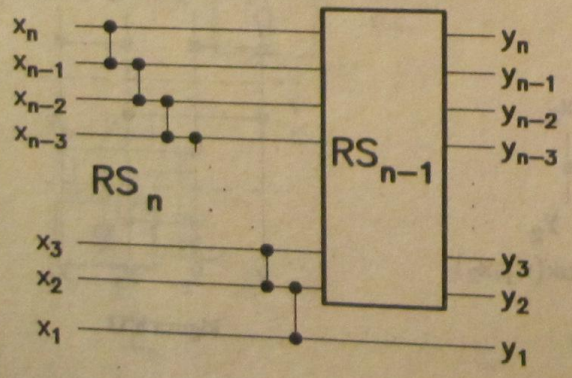



**Figura 8.12**



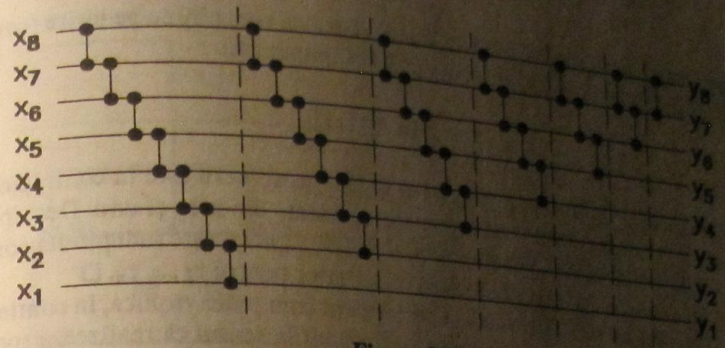



**Figura 8.13**

$$D_{RS}(2) = 1,$$
$$D_{RS}(1) = 0,$$

deci

$$D_{RS}(n) = O(n).$$

Atît complexitatea cît şi adîncimea (timpul de sortare) rezultă supărător de mari. Vom arăta în continuare că pot fi realizate reţele $RS_n$ mult mai performante.

Cum putem aprecia dacă o reţea de sortatoare (elementare) se comportă ca o reţea de sortare? *Principiul zero-unu* ne va ajuta în acest sens.

**Propoziţia 8.4** Dacă o reţea de sortatoare se comportă ca un sortator pentru $x_i \in \{0, 1\}$, $i = 1, 2, ..., n$, atunci se va comporta ca un sortator pentru orice valori ale lui $x_i$. (Principiul zero-unu) □

*Demonstraţie*

**Propoziţia 8.5** Dacă o funcţie $f(x)$ este monotonă (pentru orice $a \le b$ este îndeplinită condiţia ca $f(a) \le f(b)$) şi dacă o reţea de sortatoare generează pe ieşire $y_1, ..., y_n$ pentru intrările $x_1, ..., x_n$, atunci pentru intrările $f(x_1), ..., f(x_n)$ va genera pe ieşire $f(y_1), ..., f(y_n)$. □

*Demonstraţie* Dacă între elementele şirului de numere $x_1, ..., x_n$ există anumite relaţii de ordonare, ele vor fi conservate între elementele şirului $f(x_1), ..., f(x_n)$, conform definiţiei funcţiei $f(x)$. Conform funcţiilor de transfer ale unei reţele de sortatoare (*Definiţia 8.4*) un număr $x_i$ care se propagă de la intrarea $i$ la ieşirea $j$ va determina ca şi $f(x_i)$ de la intrarea $i$, să se propage tot la ieşirea $j$, sub forma lui $f(y_j)$, deoarece funcţiile *min* şi *max* dirijează prin reţea valorile $f(x_i)$ ca şi pe $x_i$, datorită monotoniei funcţiei $f$. □

Vom considera funcţia monotonă

$$f_a(x) = \begin{cases} 1 & x \ge a \\ 0 & altfel \end{cases}$$

care transformă orice şir de numere într-un şir de 0 şi 1, şi o reţea de sortatoare care se comportă ca un sortator pentru o secvenţă de $n$ numere $x_i$ cu $x_i \in \{0, 1\}$. Aplicăm acestei reţele, la intrare, o secvenţă $x_1, ..., x_n$ de numere oarecare şi *presupunem* că pentru *unii* $i$ ($1 \le i < n$) rezultă la ieşire $y_i > y_{i+1}$ (adică şirul nu este

sortat). Vom aplica apoi pe intrări $f_a(x_1), ..., f_a(x_n)$ şi vom obţine pe ieşire (conform *Propoziţiei 8.5*) secvenţa $f_a(y_1), ..., f_a(y_n)$. Dacă, impunem:

$$a = \frac{1}{2}(y_i + y_{i+1}),$$

atunci rezultă că $f_a(y_i) = 1$ iar $f_a(y_{i+1}) = 0$. Dar acest lucru este în contradicţie cu ipoteza că reţeaua sortează toate secvenţele formate din zero şi unu. Deci ipoteza că $y_i > y_{i+1}$ este greşită şi, pentru orice $i$ $(i = 1, 2, ..., n-1)$, este îndeplinită condiţia ca $y_i < y_{i+1}$, reţeaua de sortatoare fiind un sortator pentru $x_1, ..., x_n$. □

Folosind *principiul zero-unu* ca instrument vom putea explica, în continuare, modul de funcţionare al unui sortator performant în sensul că realizează sortarea cu o reţea de sortatoare mai puţin complexă şi mai puţin adîncă decît cea anterior definită.

**Propoziţia 8.6** Sortatorul Batcher, $S_n$, este definit dublu recursiv prin reprezentările din figurile 8.14 şi 8.15, unde $S_{n/2}$ reprezintă un sortator, de acelaşi tip, pentru o secvenţă de $n/2$ numere, iar $M_{n/2}$ este o reţea de sortatoare care, primind la intrare două secvenţe de cîte $n/2$ numere sortate, generează la ieşire o secvenţă de $n$ numere sortată, obţinută prin "amestecarea" (merge) celor două de la intrare. Acest tip de sortator este caracterizat prin $S(n) = O(n \log^2 n)$ şi $D(n) = O(\log^2 n)$. □

*Demonstraţie* Pentru ca $S_n$ din figura 8.14 să funcţioneze corect trebuie să arătăm că $M_{n/2}$ sortează cele două secvenţe sortate de la intrare. Vom folosi pentru aceasta principiul zero-unu. Considerăm că secvenţele $<a_1, ..., a_p>$ şi $<b_1, ..., b_p>$ sînt sortate şi formate numai din *0* şi *1* (figura 8.15). În prima secvenţă sînt $g$ zero-uri urmate de $p-g$ unu-uri, iar în cea de a doua sînt $h$ zero-uri urmate de $p-h$ unu-uri. La ieşirea reţelei de sortatoare $M_{p/2}$ din stînga vom avea zero-uri pe primele $\lceil g/2 \rceil + \lceil h/2 \rceil$ poziţii, iar la ieşirea celeilalte reţele $M_{p/2}$ vom avea zero-uri pe primele $\lfloor g/2 \rfloor + \lfloor h/2 \rfloor$ poziţii, deoarece la ieşirea acestora secvenţele sînt sortate. Ţinînd cont de modul în care au fost făcute conexiunile la intrarea celor două subreţele $M_{p/2}$, mărimea

$$Z = \lceil g/2 \rceil + \lceil h/2 \rceil - (\lfloor g/2 \rfloor + \lfloor h/2 \rfloor)$$

poate lua numai trei valori:

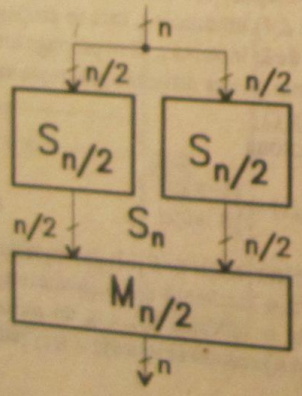



**Figura 8.14**

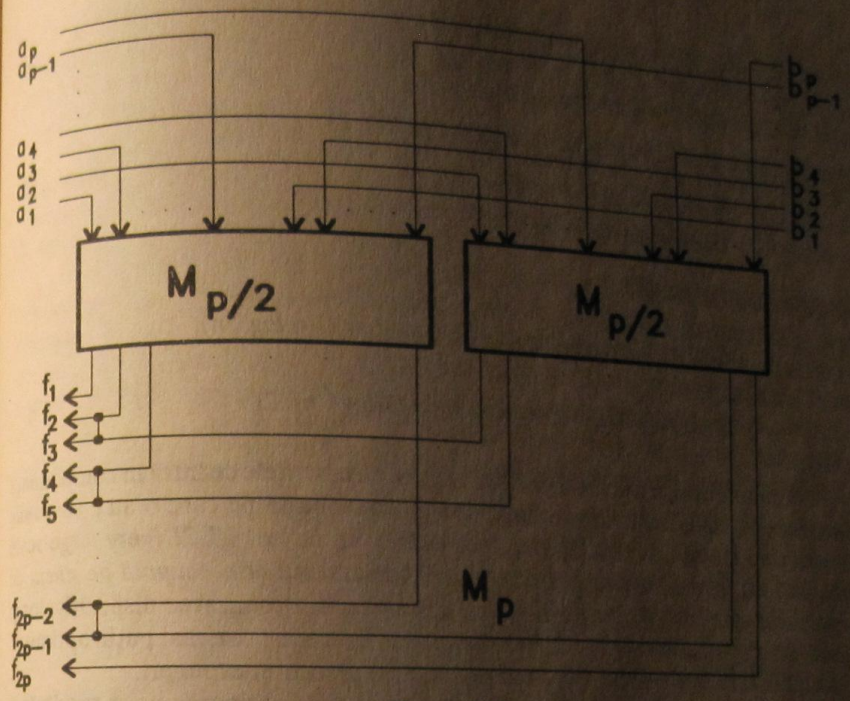



**Figura 8.15**

1. $Z = 0$ : un singur sortator, din cele $p-1$ de la ieşire, va avea zero pe intrarea inferioară (conform reprezentării din figura 8.15) şi unu pe cea superioară, cele de pe poziţii inferioare vor avea zero pe ambele intrări iar cele superioare vor avea unu pe ambele intrări; acest unic sortator va inversa poziţiile numerelor primite asigurînd o ieşire sortată ($\lfloor g/2 \rfloor + \lfloor h/2 \rfloor$) zero-uri urmate de $2p - (\lfloor g/2 \rfloor + \lfloor h/2 \rfloor)$ unu-ri).

2. $Z = 1$ : ieşirea $f_1$ este zero, primele $(\lceil g/2 \rceil + \lceil h/2 \rceil - 1)$ sortatoare vor avea pe ambele intrări zero, următoarele sortatoare vor avea pe ambele intrări unu, iar ieşirea $f_{2p}$ va fi egală cu unu; rezultă tot o secvenţă sortată.

3. $Z = 2$ : un singur sortator va avea pe intrarea superioară zero iar pe cea inferioară unu, cele de pînă la el vor avea zero pe ambele intrări, iar cele ce-i urmează vor avea unu pe ambele intrări; rezultă tot o secvenţă sortată.

Conform principiului zero-unu reţeaua de sortatoare $M_p$ generează la ieşire o secvenţă sortată, dacă cele două subsecvenţe sînt sortate.

Pentru a estima dimensiunile reţelei de sortare vom scrie întîi pentru $M_p$ :

$$s(p) = 2s(p/2) + p - 1,$$
$$s(1) = 1;$$
$$d(p) = d(p/2) + 1,$$
$$d(1) = 1.$$

Prin inducţie obţinem:

$$s(p) = p \log_2 p + 1,$$
$$d(p) = \log_2 p + 1.$$

Pentru sortatorul $S_n$ vom scrie (conform figurii 8.14):

$$S(n) = 2S(n/2) + s(n/2),$$
$$D(n) = D(n/2) + d(n/2),$$

iar prin inducţie rezultă:

$$S(n) = \frac{n}{4}(\log_2 n)(\log_2 n - 1) + n - 1 = O(n \log^2 n),$$

$$D(n) = \frac{1}{2}(\log_2 n)(\log_2 n + 1) = O(\log^2 n). \square$$

Complexitatea calculată ţine cont numai de circuitele ce intră în componenţa sortatorului. Este foarte importantă, însă, şi contribuţia pe care o au *conexiunile* atunci cînd o reţea de sortare este realizată ca un circuit *VLSI* (*very large scale integration*). Într-o realizare concretă va fi importantă *aria ocupată pe siliciu* de reţea, arie care include, pe lîngă spaţiul folosit pentru realizarea dispozitivelor şi pe cel ocupat de interconexiuni. Rezultatele vor fi, în acest caz, mai puţin optimiste. Studiul din această perspectivă depăşeşte cadrul prezentei expuneri.

Studiul teoretic al reţelelor de sortare a permis obţinerea unor rezultate şi mai performante ($S(n) = O(n \log n)$ şi $D(n) = O(\log n)$), dar aprofundarea subiectului va fi făcută numai individual de către cei interesaţi.

### 8.2.3 Reţele de permutare

**Definiţia 8.6** Un *comutator elementar* (figura 8.16*a*) este un *CLC* care primeşte la intrare două configuraţii binare, $p_1$ şi $p_2$, şi generează la ieşire, sub comanda $s$, tot două configuraţii binare, $q_1$ şi $q_2$, astfel:

- $q_1 = p_1, q_2 = p_2$ dacă $s=0$,
- $q_1 = p_2, q_2 = p_1$ dacă $s=1$. □

În reprezentări complexe intrarea de comandă nu este, de regulă, desenată. (Uneori valoarea lui $s$ este purtată chiar de configuraţia binară $p_i$ ce se propagă prin comutator.) Curent vom folosi reprezentarea din figura 8.16*b*.

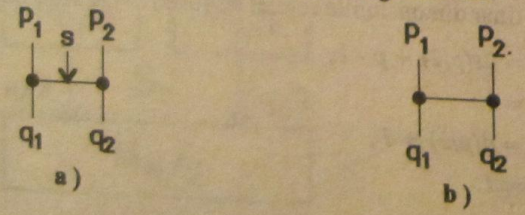



**Figura 8.16**



**Definiţia 8.7** O reţea de comutatoare elementare formează o *reţea de permutare* dacă aplicînd pe intrare secvenţa $<p_1,...,p_n>$, secvenţa $<q_1,...,q_n>$ poate reprezenta orice permutare a celei de intrare printr-o poziţionare adecvată a comutatoarelor ($s=0$ sau $s=1$). □

**Propoziţia 8.7** Reţeaua de comutatoare elementare, $RP_n$, recursiv definită în figura 8.17, reprezintă o reţea de permutare dacă cele două subreţele $RP_{n/2}$ sînt reţele de permutare. Reţeaua este caracterizată prin $S_P(n) = O(n \log n)$ şi $D(n) = O(\log n)$. (Reţeaua Waksmann) □

*Demonstraţie* Rolul comutatoarelor $K_i$, $i = 1, 2, ..., n/2$, este acela de a permite oricărui $p_i$, $i = 1, 2, ..., n$, accesul la oricare din cele două permutatoare $RP_{n/2}$. Rolul comutatoarelor $S_i$, $i = 2, 3, ..., n/2$, este de a permite ca, oricare două intrări alăturate să poată fi permutate pe ieşiri cu indici ce au aceeaşi paritate. Pentru claritate am alcătuit tabelul 8.1 în care sînt prezentate cele patru cazuri posibile. În acest tabel $i$, $j$ şi $k$ sînt numere pare (diferite de zero). Cu $s_{k/2}$ am notat poziţia comutatorului $K_{k/2}$ iar cu $s_{i/2}$ şi $s_{j/2}$ am notat poziţiile comutatoarelor $S_{i/2}$ şi $S_{j/2}$.

| Caz | $P_{k-1}$ | $P_k$ | $s_{k/2}$ | $s_{i/2}$ | $s_{j/2}$ |
|---|---|---|---|---|---|
| a. | $q_{i-1}$ | $q_{j-1}$ | 0/1 | 0/1 | 1/0 |
| b. | $q_{i-1}$ | $q_j$ | 0/1 | 0/1 | 0/1 |
| c. | $q_i$ | $q_{j-1}$ | 1/0 | 0/1 | 0/1 |
| d. | $q_i$ | $q_j$ | 1/0 | 0/1 | 1/0 |

**Tabelul 8.1**

Valorile de pe două intrări alăturate, una impară şi una pară, pot fi permutate în cele patru moduri reprezentate în tabel. Dacă paritătile ieşirilor sînt diferite, atunci îşi dovedesc utilitatea numai comutatoarele de pe intrare, deoarece există posibilitatea ca cele de pe ieşire să fie comandate cu zero (nu permută la ieşire cele două intrări). Dar în cazurile *a* şi *d* nu poate fi evitată folosirea comutatoarelor de

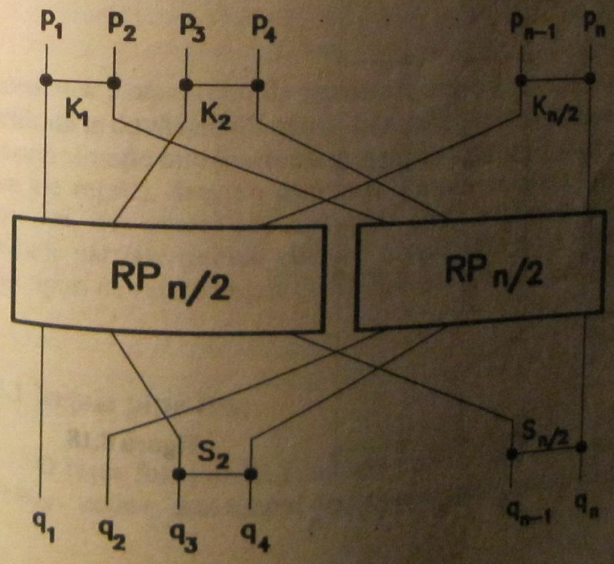



**Figura 8.17**

pe ieşire. Este situaţia în care ambele ieşiri, la care ajung cele două intrări, au indici cu aceeaşi paritate.

Pentru evaluarea cantitativă a reţelei scriem:

$$S(n) = 2S(n/2) + n - 1,$$
$$S(1) = 0,$$

deci

$$S(n) = n \log_2 n - n + 1 = O(n \log n);$$

$$D(n) = D(n/2) + 2,$$
$$D(1) = 0,$$

deci

$$D(n) = 2 \log_2 n = O(\log n). \square$$

**Exemplul 8.8** În figura 8.18 este reprezentată o reţea de permutare pentru cazul particular $n=8$. □

Ne putem întreba care este utilitatea unei astfel de reţele. O aplicaţie posibilă este aceea de a interconecta flexibil între ele un număr de $n$ procesoare. În figura 8.19 procesoarele $P_1, P_2, ..., P_n$ sînt interconectate între ele prin intermediul reţelei $RP_n$. Remarcăm faptul că pentru cele $n$ procesoare avem $S_p(n) = O(n)$, iar

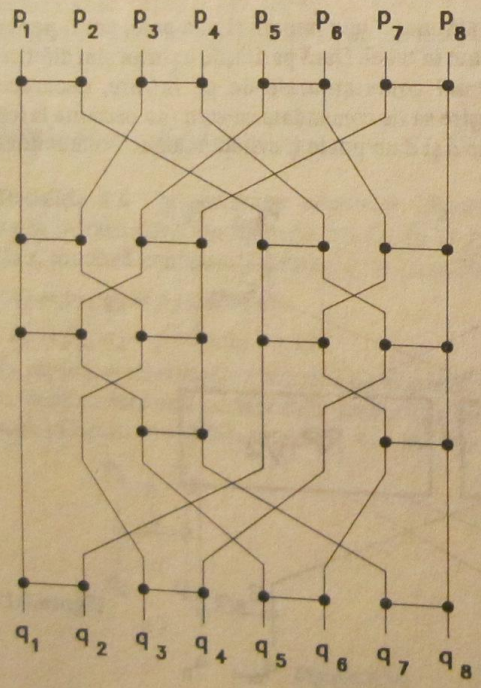



**Figura 8.18**



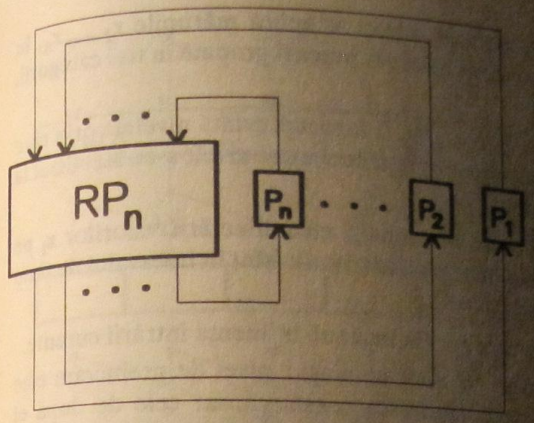



**Figura 8.19**

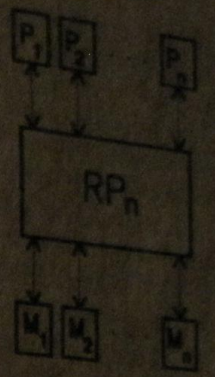



Figura 8.20

pentru reţeaua care le interconectează avem $S_{RP}(n) = O(n \log n)$. Va trebui să ne obişnuim cu ideea că interconectarea poate costa mai mult decît procesarea propriuzisă în reţelele complexe de procesoare. O altă aplicaţie ar fi cea prin care se interconectează flexibil $n$ procesoare la $n$ module de memorie ($M_i$), ca în figura 8.20. În acest caz se poate folosi proprietatea comutatoarelor, realizate cu tranzistoare *MOS*, de a fi bidirecţionale (paragraful 4.3.2). Acest mod de legare asigură un acces multiplu la o memorie văzută ca suma celor $n$ module de memorie şi, în acelaşi timp, asigură şi interconectarea procesoarelor între ele prin intermediul memoriilor.

## 8.3 Reţele cu memorie

Necesitatea elementelor de memorare într-o reţea funcţională complexă poate fi justificată în două situaţii opuse. Cantitatea de date de care dispune sistemul într-o anumită situaţie este foarte mare şi poate fi distribuită spre procesare în mai multe zone ale reţelei. Sau, nu dispunem sincron de toate datele pe care dorim să le procesăm, dar numai ansamblul lor poate împlini procesarea. Atunci date sau rezultate parţiale trebuie găzduite temporar în reţea. Vom studia, în consecinţă, două tipuri de reţele: reţele de tip *pipe-line* şi reţele *sistolice*.

### 8.3.1 Reţele pipe-line

O reţea funcţională complexă cu $m$ niveluri, are asociat un timp de calcul $T = mt_p$, unde $t_p$ este timpul de propagare printr-un nivel. Presupunînd că $t_p$ este

cunoscut şi că în momentul $t_1$ la intrarea reţelei se aplică mărimile $x_1, ..., x_n$, în intervalul de timp $[t_1, t_1 + T]$, nivelele reţelei vor putea fi grupate în trei categorii, în funcţie de $t_2 \in [t_1, t_1 + T]$:

1. primele $l$ nivele, unde $l = \lfloor (t_2 - t_1)/t_p \rfloor$, care reprezintă nivelele prin care efectul configuraţiilor binare $x_i$ s-a propagat; ieşirile acestor nivele s-au stabilizat la valoarea finală, corespunzător intrării;

2. nivelul $l+1$ prin care tocmai se propagă efectul setării valorilor $x_i$ pe intrare la $t_1$; ieşirile acestui nivel sînt în regim tranzitoriu (sînt în intervalul de timp în care se manifestă fenomenele de hazard);

3. ultimele $m - (l+1)$ nivele care nu sînt încă sub influenţa intrării curente.

Rezultă că în fiecare moment cel mult un singur nivel de prelucrare este "activ", cele dinaintea lui menţinîndu-i intrarea constantă, iar cele de după el nefiind încă active. Efectul secvenţei $<x_1, ..., x_n>$ se propagă prin reţea depăşind după fiecare $t_p$ cîte un nivel de prelucrare. Dacă $n$ este suficient de mare putem spune că imensa majoritate a reţelei nu este activă, iar la limită, că de fapt reţeaua este practic inactivă. Deci, cum am putea face ca reţeaua să fie activată într-o cît mai mare măsură?

Discuţia anterioară a fost făcută în ipoteza că $t_p$ este o valoare bine cunoscută şi reproductibilă tehnologic. După cum bine ştim, acest lucru nu este adevărat. Nu se poate controla riguros propagarea semnalelor logice printr-o reţea combinaţională. Din acest motiv va trebui să introducem structuri suplimentare care să permită controlul propagării prin reţea, astfel încît să putem folosi disponibilitatea funcţională oferită de nivelele "inactive". Ateţie! Dacă avem ce face cu această disponibilitate!

Vom expune *principiul pipe-line de structurare* pe un exemplu.

**Exemplul 8.9** În figura 8.21 este reprezentată o reţea de calcul al prefixelor în varianta pipe-line. Faţă de structura anterior discutată (figura 8.6) au fost adăugate, în plus, patru rînduri de cîte opt latch-uri, fiecare latch putînd stoca o variabilă $x_i$. Rîndurile succesive de latch-uri sînt comandate de semnale de ceas în antifază, astfel încît *propagarea semnalelor prin reţea este controlată*, în virtutea principiului master-slave. Într-adevăr:

- în intervalul $T_1$ (figura 8.22) se încarcă primul nivel de latch-uri şi este activ primul nivel de circuite;

- în semiperioada $T_2$ ieşirea primului nivel combinaţional este vărsată în cel de al doilea nivel de latch-uri şi cel de al doilea nivel combinaţional este activat; primul nivel de latch-uri este închis astfel încît conţinutul acestora nu se modifică;

- în $T_3$ rezultatul prelucrării pe nivelul secund se înscrie în al treilea nivel de latch-uri, se blochează al doilea nivel de memorare şi se deschide primul pentru a se înregistra o nouă secvenţă de intrare, care este prelucrată pe primul nivel logic;

- $T_4$ este semiperioada în care rezultatul primei secvenţe este înscris în ultimul nivel de latch-uri, iar rezultatul parţial, asociat secvenţei secunde, se înscrie în cel de al doilea nivel de memorare.



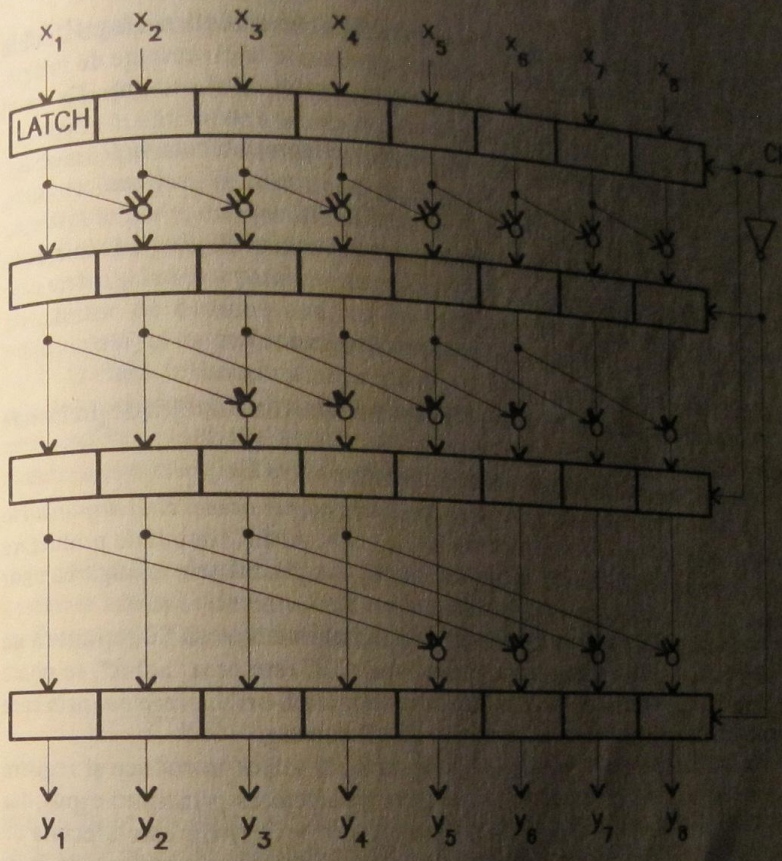
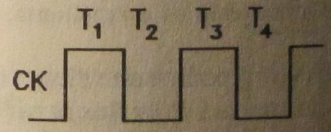



Figura 8.21



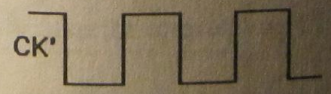




Figura 8.22

Astfel, după primele două cicluri de ceas un prim rezultat este accesibil la ieşire, iar unul parţial se află în interiorul reţelei. *La fiecare ciclu de ceas*, în continuare, dacă intrarea este alimentată corespunzător cu noi secvenţe, *se va obţine la ieşire un rezultat.*

Reţeaua din acest exemplu prelucrează simultan două secvenţe de tipul $<x_1, ..., x_n>$, generînd în fiecare ciclu cîte un rezultat sub forma unei secvenţe de forma $<y_1, ..., y_n>$, cu condiţia ca la intrare să fie realimentată în fiecare ciclu de ceas. □

În reţeaua pipe-line anterior exemplificată se realizează un "dublu paralelism". Odată, sînt prelucrate în paralel elementele unei secvenţe de intrare, $<x_1, ..., x_n>$, iar a doua oară, sînt prelucrate în paralel două secvenţe (în cazul general mai multe: $\lceil 0{,}5 \log_2 n \rceil$) de intrare.

Timpul de calcul pentru $p$ seturi de cîte $n$ configuraţii binare va fi

$$T(n, p) = t_c(\log_2 n + p - 1),$$

unde $t_c$ este perioada ceasului. Timpul mediu asociat unei configuraţii de intrare va fi, pentru $p$ suficient de mare

$$T(n) = \frac{1}{p} T(n, p) \approx t_c = const.$$

**Propoziţia 8.8** O reţea de tip pipe-line poate *tinde* către o reţea de prelucrare caracterizată prin $D(n) = O(1)$. □

Calculul anterior justifică această propoziţie.

*Utilitatea unei structurări de tip pipe-line apare numai atunci cînd dispunem la intrare de un flux ritmic de secvenţe* de tip $<x_1, ..., x_n>$. Altfel, timpul de prelucrare va fi tot $O(f(n)) \neq O(1)$, dar într-o structură inutil complicată prin adăugarea unor latch-uri.

Acest mod de structurare a unui sistem digital este bine să fie reţinut ca un procedeu cu aplicabilitate mai largă. Oriunde un *CLC* este prea "adînc", se poate dovedi utilă fragmentarea lui prin intercalarea unor latch-uri *sau registre*, prin care se poate accelera ritmul în care fluxurile de date îl parcurg.

În locul latch-urilor folosite (ca în figura 8.21) se pot introduce şi registre (realizate cu structuri master-slave). Acest lucru nu afectează principiul expus, dar uneori complică inutil structura. Convergenţa $D(n) \to O(1)$ nu este afectată de opţiunea între latch-uri sau registre. Această opţiune poate afecta cel mult constantele, fapt ce nu ne interesează în acest capitol. Prin registre, gradul de utilizare al nivelelor din reţea se dublează (se înmulţeşte cu o constantă, deci se obţine un efect "neesenţial").

Reţinem că opţiunea pentru o reţea cu sau fără pipe-line este dictată de tipul de aplicaţie. Dacă, reţeaua este măcar temporar alimentată de un flux ritmic de date (la fiecare perioadă a ceasului), atunci este justificată opţiunea pentru o reţea cu facilităţi de memorare.

Remarcăm, în final, că o reţea de tip pipe-line este formată din două subreţele:

1. o reţea de circuite combinaţionale ce asigură prelucrarea propriuzisă;
2. o reţea de elemente de memorare;

astfel încît, la limită, fiecare configuraţie binară stocată în reţea este afiliată la un element de prelucrare după un algoritm ce ţine de funcţia reţelei (calculul prefixelor, sortarea, permutarea ş.a.).



### 8.3.2 Reţele sistolice de ordinul unu

*Reţelele sistolice* au fost promovate, în principal, prin lucrările lui Kung şi Leiserson în efortul de a face ca funcţiuni realizate la nivel software să migreze către hardware, pentru creşterea performanţelor cu care ele se realizează. Odată ce anumite funcţiuni se conturează ca deosebit de utile, prin frecvenţa şi efectul cu care sînt folosite, este justificată încercarea de a le implementa în sisteme cu ordin cît mai scăzut. Un sistem programabil are cel puţin ordinul patru, dar, după cum vom vedea şi în acest paragraf, anumite funcţiuni se vor putea implementa la nivelul sistemelor de ordinul unu, cu un efort structural acceptabil (complexitate polinomială) şi cu performanţe de viteză excepţionale.

O clasă de funcţiuni numerice care se detaşează net este cea care se referă la calculul vectorial şi matricial. Au fost concepute, în consecinţă, structuri sistolice care sprijină evaluarea acestor categorii de funcţiuni. După cum ştim, în acest domeniu este esenţială evaluarea unor expresii aritmetice de forma

$$c = a_1 b_1 + a_2 b_2 + ... + a_n b_n.$$

Modulul elementar, care poate fi folosit în structurarea unor reţele sistolice destinate acestui domeniu, este reprezentat în figura 8.23, unde $S$ este un sumator iar $P$ este un *CLC* care realizează produsul a două numere. Rezultă că

$$A' = A,$$
$$B' = B,$$
$$C' = A \times B + C,$$

ieşirile fiind condiţionate de ceasul *CK*, al cărui palier pozitiv face transparent latch-ul de la ieşire. (Simbolurile $\times$ şi $+$ au semnificaţie aritmetică.) În figura 8.24 este reprezentat simbolul logic asociat acestui sistem de ordinul unu.

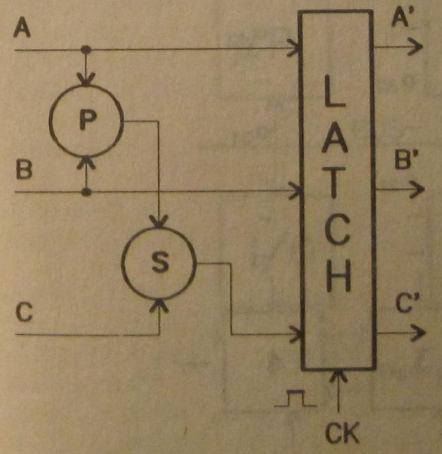



**Figura 8.23**

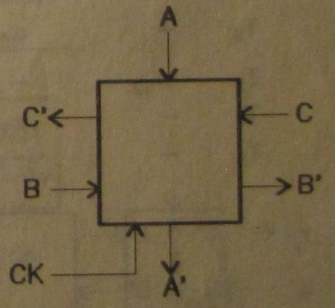



**Figura 8.24**

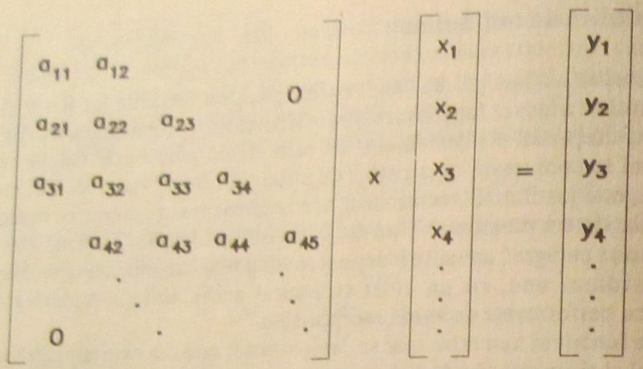


Figura 8.25

**Exemplul 8.10** Vom prezenta, în continuare, un sistem sistolic ce înmulţeşte o matrice de tip bandă cu un vector, conform reprezentării din figura 8.25. Elementele vectorului rezultat vor fi:

$$y_1 = a_{11}x_1 + a_{12}x_2$$
$$y_2 = a_{21}x_1 + a_{22}x_2 + a_{23}x_3$$
$$y_3 = a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4$$
$$y_4 = a_{42}x_2 + a_{43}x_3 + a_{44}x_4 + a_{45}x_5$$
$$y_5 = a_{53}x_3 + \ldots$$
$$\vdots$$

Structura fizică şi modul în care aceasta este ritmic (sistolic) alimentată, sînt prezentate în figura 8.26. Sincron cu fiecare ciclu de ceas (figura 8.27), elementele

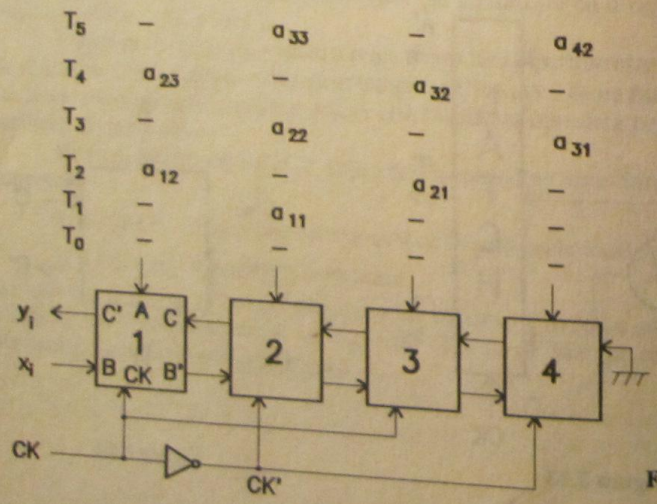



Figura 8.26



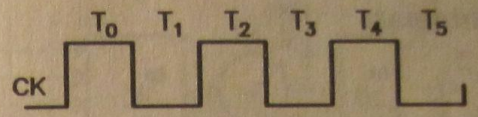



Figura 8.27

vectorului $x$ sînt aduse la intrarea $B$ a primului modul. Ceasul aplicat în antifază, modulelor numerotate par şi impar, permite realizarea unor structuri master-slave din latch-urile modulelor succesive, asigurînd astfel propagarea controlată a mărimilor prin sistem.

La $T_0$ (figura 8.27) $x_i = x_1$ se înscrie în modulul *1*.

La $T_1$ la intrările modulului *2* avem $B = x_1$ şi $A = a_{11}$ şi va rezulta ieşirea $C' = a_{11}x_1$ şi $B' = x_1$ (asigurînd propagarea lui $x_1$ spre dreapta pentru ca în ciclul următor să se întîlnească cu $a_{21}$ ş.a.m.d.).

La $T_2$ pe intrările modulului unu avem $A = a_{12}$, $B = x_2$, $C = a_{11}x_1$ şi va rezulta $C' = a_{11}x_1 + a_{12}x_2 = y_1$, $B' = x_2$.

În semiperioada $T_4$, în urma unui proces similar va apare $y_2$ ş.a.m.d. □

Înmulţirea a două matrici se face cu o reţea bidimensională de module. (Se poate consulta, în acest sens, bibliografia.) Din acest moment, realizarea unor sisteme pentru funcţia de convoluţie sau pentru calculul transformatelor Fourier este imediată. Sistemele sistolice pot funcţiona în calitate de *coprocesoare* pe bus-ul unui sistem de calcul (figura 8.28), fiind apelate ca sisteme periferice către care se transferă operanzii unor funcţiuni critice şi de la care se recepţionează şirul ce reprezintă rezultatul.

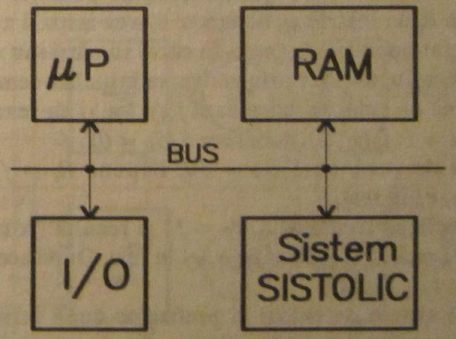



Figura 8.28

## 8.4 Reţele de automate

Nodurile unei reţele complexe pot fi formate şi din automate finite elementare (sisteme de ordinul doi). Optimizarea raportului dintre complexitatea reţelei şi timpul de prelucrare asociat, impune uneori astfel de soluţii. Un exemplu posibil este cel al *memoriei conexe* (paragraful 5.8.3) care îşi poate găsi implementarea ca o reţea dee automate finite elementare.

### 8.4.1 RCP ca reţea de automate

Vom relua numai unul din exemplele tratate anterior ca reţea de ordinul zero (combinaţională) şi ca reţea de ordinul unu (cu memorie): reţeaua de calcul al prefixelor. Dacă nu dispunem de fluxuri de date care să justifice o reţea pipe-line pentru calculul prefixelor, atunci, ar fi utilă conceperea unei reţele care să dispună numai de $n$ *CLC*-uri ce realizează funcţia $f = f_2$, deoarece într-o reţea combinaţională vor fi "active" cel mult $n-1$ odată.

**Exemplul 8.11** Vom realiza pentru $n=8$ circuitul pentru calculul prefixelor pornind de la soluţia dată în *Exemplul 8.4*. Dacă dorim să folosim numai $n$ circuite de tip $f_2$, atunci, va trebui să "refolosim" de mai multe ori acelaşi circuit $f_2$ în diverse etape ale calculului. "Refolosirea" implică o buclă. Mai clar sînt redate lucrurile în figura 8.29, unde un operand al funcţiei $f_2$ vine direct dintr-un registru $R_i$ $(i = 1, 2, ..., 8)$ iar celălalt este selectat printr-un multiplexor $M_i$ de la intrările $x_1, ..., x_n$ sau de la ieşirile registrelor în care se află rezultatele parţiale, corespunzătoare calculului "pe nivelul anterior".

În primul ciclu $S_1S_0 = 00$ (semnalele de selecţie comune tuturor multiplexoarelor) şi în $R_i$ se înscrie $x_i$, deoarece starea iniţială a fiecărui $R_i$ este $e$, elementul neutru asociat operaţiei $f_2$ (zero, în cazul sumării sau unu logic, în cazul funcţiei *AND*, spre exemplu, obţinut prin activarea iniţială a semnalului *RESET*).

În ciclul al doilea de ceas, se calculează $f_2(y_i, y_{i-1})$ cu rezultat în $R_i$, pentru $i = 2, ..., 8$, iar pentru $i = 1, f_2(y_i, e)$, deoarece $S_1S_0 = 01$.

Al treilea ciclu de ceas, cu $S_1S_0 = 10$, impune $R_i = f_2(y_i, y_{i-2})$ pentru $i = 3, ..., 8$ şi $R_i = f_2(y_i, e)$ în rest.

În al patrulea ciclu se comandă $S_1S_0 = 11$ şi rezultă (conform schemei) că $R_i = f_2(y_i, y_{i-4})$ pentru $i = 5, ..., 8$ şi $R_i = f_2(y_i, e)$ în rest. După acest ciclu rezultatul este definitiv la ieşirile $y_1, ..., y_n$. □

Realizarea unui sistem de calcul al prefixelor după principiul expus prin exemplul anterior generează o structură cu $S(n) = O(n \log n)$, ce are asociat un timp de calcul $T(n) = O(\log n)$. Observăm că în evaluarea complexităţii efectul dominant este dat de multiplexoare, restul structurii avînd complexitatea $O(n)$. Reţeaua se poate dovedi utilă dacă *CLC*-urile asociate funcţiilor $f_2$ sînt relativ complexe. Astfel, pentru un $f_2$ particularizat prin funcţia de sumare, această soluţie se dovedeşte foarte utilă deoarece constanta asociată multiplexoarelor în evaluarea



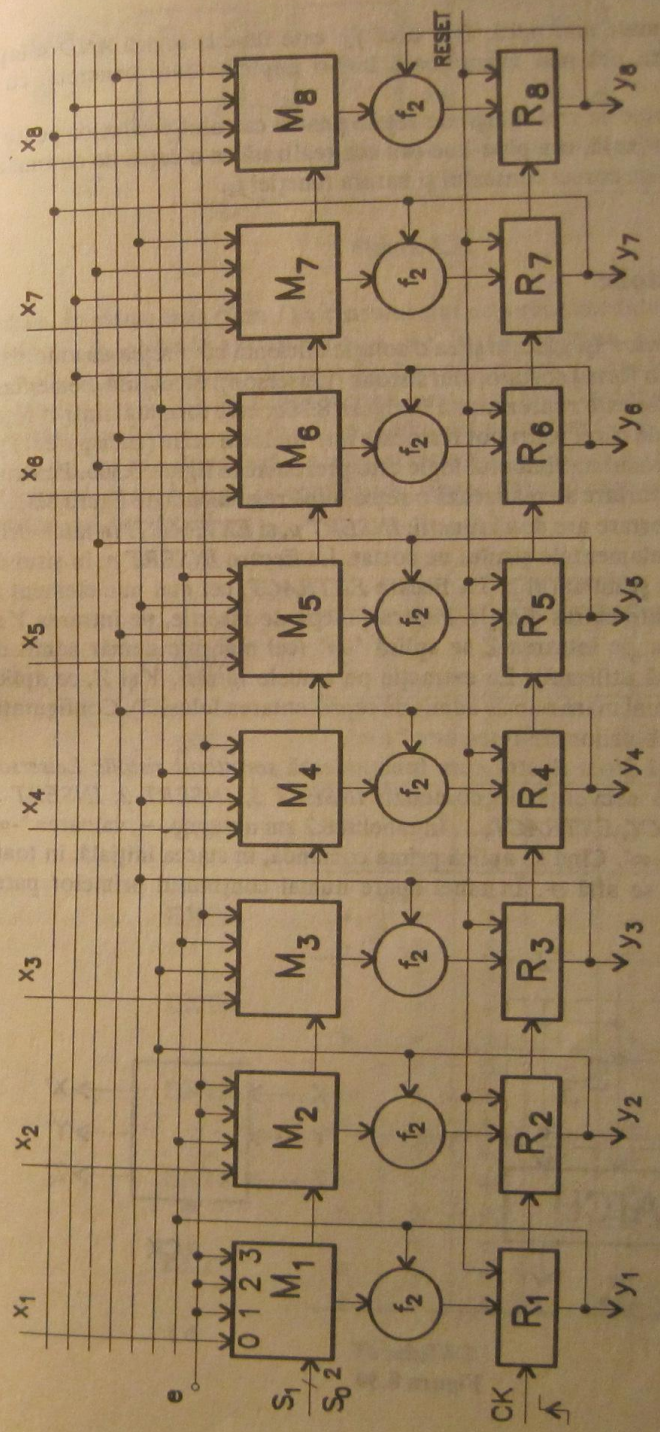



Figura 8.29

complexităţii este mult mai mică. Dar dacă $f_2$ este funcţia logică *AND* soluţia combinaţională este net mai avantajoasă într-o implementare concretă, cu $n$ limitat.

Oricare dintre cele trei soluţii de reţele pentru calculul prefixelor poate fi utilă, cea combinaţională, cea pipe-line sau cea realizată ca o reţea de automate. Totul este să analizăm corect contextul şi natura funcţiei $f_2$.

### 8.4.2 Sortator sistolic

*Funcţia de sortare* îşi găseşte şi ea o soluţie eficientă cu o reţea de automate finite elementare sub forma sortatorului sistolic (Leiserson). Structura elementară de la care se porneşte este reprezentată în figura 8.30*a*; este formată dintr-o reţea de sortare cu $n=3$, ale cărei ieşiri pot fi stocate într-un latch activ (transparent) pe palierul pozitiv al ceasului. Simbolul logic este prezentat în figura 8.30*b*. Pentru a forma un circuit de sortare se realizează o reţea după regula dată de figura 8.31.

Reţeaua de sortare are două funcţii: *INSERT p*, şi *EXTRACT*. În latch-urile reţelei sînt stocate elementele şirului de sortat. La fiecare *INSERT p*, în şirul de sortat este introdus şi numărul $p$. La fiecare *EXTRACT*, cel mai mic element al şirului de numere este extras din şir. Pentru funcţia de inserţie, pe intrarea $Y$ se aplică numărul $p$ iar pe intrarea $Z$ se aplică "$-\infty$" (cel mai mic număr admis de reprezentarea binară utilizată). La extracţie pe ambele intrări, $Y$ şi $Z$, se aplică numărul "$+\infty$" (cel mai mare număr admis de reprezentarea folosită). Configuraţia binară iniţială a latch-urilor din reţea este "$+\infty$".

**Exemplul 8.11** Vom ilustra cum funcţionează *sortatorul sistolic Leiserson* pentru următoarea secvenţă de comenzi: *INSERT 5, INSERT 2, INSERT 3, EXTRACT, EXTRACT, EXTRACT,...* . În tabelul 8.2 am notat cu $-$, valoarea "$-\infty$" şi cu $+$, valoarea "$+\infty$". Cînd se aplică prima comandă, în starea iniţială, în toate celulele de stocare se află $+$. În tabel apare numai conţinutul primelor patru

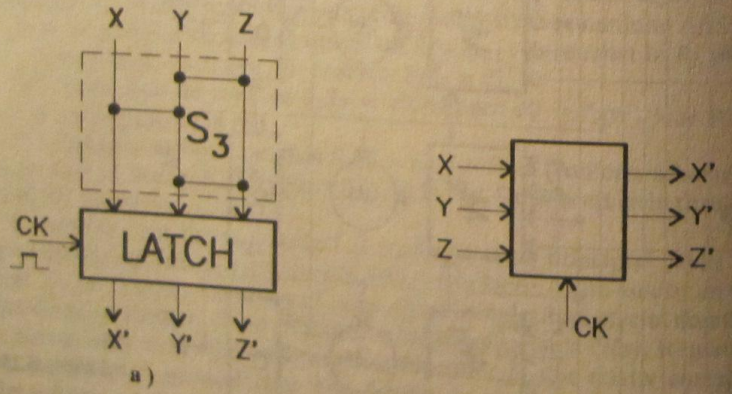



**Figura 8.30**



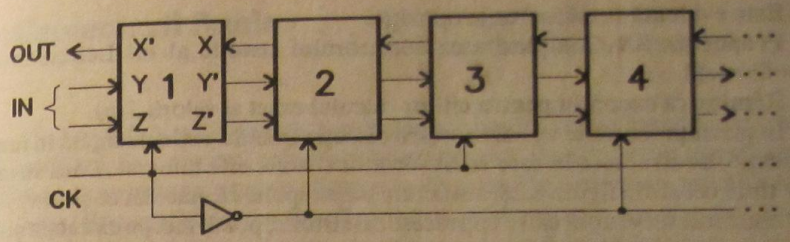



**Figura 8.31**

module. În prima fază (faza *1*) a ceasului sînt acţionate modulele impare şi în *1* se înscrie $(-, 5, +)$ iar în al treilea $(+, +, +)$. În faza a doua a ceasului (faza *0*, cu $CK = 0$), în funcţie de modificarea din faza anterioară, în *2* se înscrie $(5, +, +)$ iar în *4* rămîne $(+, +, +)$. Acestei noi configuraţii i se aplică a doua comandă ce aduce pe intrări $Y = 2$, $Z = -$. În faza *1* în modulul *1* se înscrie $(-, 2, 5)$ şi în *3* se înscrie $(+, +, +)$, iar în a doua fază a aceluiaşi ciclu de ceas $(2, 5, +)$ în *2* şi $(+, +, +)$ în *4*. Cititorul poate lectura în continuare, singur, tabelul care descrie funcţionarea sortatorului sistolic. □

| | I | 1 | 0 | 1 | 0 | |
|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | |
| | | + | + | + | + | + |
| INS 5 | 5 | + | + | + | + | |
| | − | + | + | + | + | |
| | | − | 5 | + | + | + |
| INS 2 | 2 | 5 | + | + | + | |
| | − | + | + | + | + | |
| | | − | 2 | + | + | + |
| INS 3 | 3 | 2 | 5 | + | + | |
| | − | 5 | + | + | + | |
| | | − | 2 | 5 | + | + |
| EXTR | + | 2 | 3 | + | + | |
| | + | 3 | 5 | + | + | |
| | | 2 | 3 | 3 | 5 | + |
| EXTR | + | + | + | 5 | + | |
| | + | + | + | + | + | |
| | | 3 | 5 | 5 | + | + |
| EXTR | + | + | + | + | + | |
| | + | + | + | + | + | |
| | | 5 | + | + | + | |
| EXTR | + | + | + | + | + | |
| | + | + | + | + | + | |
| | | + | + | + | + | |
| EXTR | + | + | + | + | + | |
| | + | + | + | + | + | |

**Tabelul 8.2**

Este evidentă următoarea propoziţie.

**Propoziţia 8.9** Complexitatea sortatorului sistolic al lui Leiserson este $S(n) = O(n)$. □

Rămîne ca exerciţiu pentru cititor calculul exact al valorii $S(n)$.

În privinţa timpului asociat sortării discuţia trebuie puţin nuanţată în funcţie de aplicaţie, de structura în care acest sortator sistolic este integrat. Dacă sistemul este de tipul celui din figura 8.28, atunci am putea spune că, imediat ce şirul $x_1,..., x_n$ a fost transferat secvenţial către coprocesorul sistolic, poate începe extracţia şirului $y_1,..., y_n$, ce reprezintă rezultatul sortării. Aşa privite lucrurile, tot timpul este consumat cu transferurile într-un interval $T(n) = O(n)$, iar pentru sortarea propriuzisă timpul consumat este $T(n) = O(1)$. Dacă dispunem într-un registru de întregul şir $x_1,..., x_n$ şi se pune problema serializării lui, atunci într-adevăr $T(n) = O(n)$. Dar aceasta este o situaţie ideală pentru un $n$ oricît de mare. Deci teoretic $T(n) = O(n)$, dar practic $T(n) = O(1)$, pentru că problema transferului pe un canal oricît de larg, dar finit, este $T(n) = O(n)$. Nu putem cere ca şirul să fie sortat înainte de a fi rostit în întregime!

- **Sortatorul sistolic este structurat echilibrat** la nivelul sistemelor de ordinul doi. Această afirmaţie este foarte încurajatoare pentru viitorul sistemelor digitale. Într-adevăr, dacă detaliem două celule elementare succesive din sortatorul sistolic, rezultă reprezentarea din figura 8.32, în care se pun în evidenţă cele două latch-uri comandate în antifază, ce asigură închiderea buclei în condiţii de stabilitate şi de neblocare a structurii. Între cele două latch-uri se află *CLC*-urile formate de $S_3$ şi remarcăm faptul că şi între primul şi cel de al doilea latch dar şi între cel de al doilea şi primul latch se regăseşte acelaşi circuit combinaţional. Nu ne aflăm în situaţia de puternică dezechilibrare, anterior evidenţiată în figura 7.3*b*.

Recuperarea pentru spaţiul sistemelor echilibrat structurate a încă unui nivel este un fapt deosebit de pozitiv pentru evoluţia sistemelor digitale. Acest exemplu din clasa sistemelor de ordinul doi ne face să ne gîndim că vom putea să construim şi alte reţele funcţionale în care să recuperăm ceea ce am pierdut structurînd neechilibrat în graba după efecte imediate sau în tendinţa de a ne supune restricţiilor tehnologice. O tehnologie restrictivă şi o viziune funcţională insuficient limpezită ne-au adus în preajma sistemelor centrate pe microprocesoare. Depăşirea acestui stadiu este posibilă dacă şi mecanismele de structurare pe care le vom practica vor deveni din ce în ce mai echilibrate.

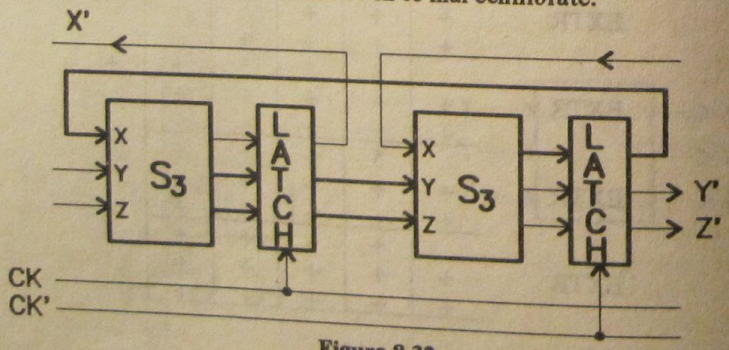



**Figura 8.32**



## 8.5 Comentarii finale

1. Apariţia şi proliferarea preocupărilor pentru reţele funcţionale complexe este facilitată de *progresul tehnologiilor electronice*, care permite abordarea unor probleme ce anterior nu puteau fi puse în discuţie. De asemenea, practica folosirii calculatoarelor, multitudinea de aplicaţii deja implementate, au permis *evidenţierea unor funcţiuni esenţiale* şi prin aceasta critice. Aceste funcţiuni au fost, şi sînt încă, realizate prin programe, cu o eficienţă limitată de performanţele actuale ale sistemelor de calcul. Acestea, în calitate de maşini universale, acoperă tot spectrul funcţiilor calculabile dar nu pot atinge performanţele cerute de anumite situaţii sau aplicaţii limită. Se întrevede însă posibilitatea unei *migrări funcţionale* dinspre software către hardware. O formă particulară a acestui proces de migrare o constituie reţelele funcţionale complexe, care pot realiza în condiţii de mare performanţă funcţiuni de mare utilitate într-un sistem.

Perioada pe care o parcurgem şi cele ce vor urma, vor fi marcate de preocuparea de a evidenţia funcţiuni sau seturi de funcţiuni (arhitecturi) ce îşi vor putea găsi implementarea sub forma unor reţele funcţionale. Complexitatea acestor reţele va putea creşte odată cu progresele tehnologice stimulate chiar de acest nou mod de abordare.

2. Aria pe care o ocupă o reţea complexă pe siliciu este o problemă ce nu a fost abordată în acest curs. Analiza din perspectiva implementării prin tehnologii *VLSI* pune în evidenţă rolul, uneori determinant, al *interconexiunilor*. Nu de puţine ori acestea pot determina schimbarea funcţiei ce leagă complexitatea circuitului de $n$. O rediscutare din această perspectivă se va impune atunci cînd reţelele vor fi privite ca circuite *VLSI*. Preocuparea pentru a asigura conexiuni cît mai puţine şi mai scurte poate deveni uneori dominantă atrăgînd după sine asumarea unei complexităţi sporite a circuitelor, dacă aceasta are drept consecinţă minimizarea ariei ocupate de interconexiuni.

Am fost obişnuiţi să gîndim circuitele electronice neglijînd conexiunile. Nu trebuie să ajungem în situaţia în care rolul acestora să fie mai important decît al circuitelor pe care le leagă. Comunicarea în sistemele fizice naturale nu susţine un astfel de model. Sistemele naturale sînt mult mai *conexe*, presupunînd o interconectare mult mai directă şi mai puţin costisitoare. Deci atenţie la fire!

3. Aproape toate reţelele studiate au fost caracterizate de o adîncime $D(n) = O(\log n)$, iar din partea cercetătorilor nu se observă tendinţa de a scade suplimentar această dimensiune. Merită să obţinem o adîncime sub una logaritmică? Condiţiile oferite de actualele tehnologii electronice ne determină să răspundem: nu. Acest răspuns este justificat prin faptul că accesul la şi de la reţea al elementelor şirurilor $<x_1,..., x_n>$ şi $<y_1,..., y_n>$ nu se poate face, în actualul context tehnologic, decît într-un interval de timp $T(n) = O(n)$. Nu putem imagina un canal pe care să poată fi transferaţi un număr oricît de mare de biţi. Oricît de larg este dimensionat un canal, dimensiunea lui este constantă, deci $T(n) = c \cdot n$ şi oricît de mic ar fi $c$ (pentru canale foarte largi), timpul asociat transferurilor rămîne tot proporţional cu $n$.

Dacă vom socoti că evaluarea unei funcţii într-o reţea se face într-un timp ce include şi timpul de transfer al secvenţei de intrare iar apoi al secvenţei de ieşire, atunci nu are sens să ne grăbim cu evaluarea pentru a obţine minimizarea unui timp neglijabil. Dacă transferul presupune $T_T(n) = O(n)$ iar prelucrarea $T_P(n = O(\log n)$, atunci $T_T(n) + T_P(n) = O(n)$ deci, nu are sens să-l micşorăm pe $T_P(n)$.

Poate nu are sens nici să ajungem la $T_P(n) = O(\log n)$. Merită totuşi să facem acest efort deoarece, în *aplicaţii concrete cu n limitat*, constantele asociate lui $T_P(n)$ pot fi mult mai mari decît cele asociate lui $T_T(n)$. De asemenea, uneori înlănţuim direct reţele funcţionale realizînd un şir de prelucrări, astfel încît timpul total asociat acestui proces poate deveni $T_T(n) + T_{P1}(n) + ... + T_{Pm}(n)$. Din jocul constantelor, în situaţii concrete, $T_T(n)$ poate să piardă ponderea principală, deci merită să gîndim reţele cu adîncime logaritmică.

Odată cu apariţia unor tehnologii în care să fie posibil ca $T_T(n) = O(1)$, se va putea pune problema unor $T_P(n)$ cu o dependenţă mai favorabilă. Tehnologiile optice se pare că vor oferi astfel de perspective. Pînă atunci, atingerea unui timp de prelucrare logaritmic va constitui un ţel care, odată atins, atenuează încrîncenarea cu care încercăm găsirea unor soluţii mai bune.

4. Mai există şi un alt motiv pentru care procesul de minimizare al adîncimii unei reţele este stopat. Trebuie să ne punem şi problema preţului pe care-l plătim pentru scăderea lui $D(n)$. Un raţionament rudimentar ne arată că odată cu scăderea lui $D(n)$ se produce o creştere a lui $S(n)$. Deci, preţul plătit este evident. Problema care se pune este: cît de mare poate fi acest preţ? Să analizăm cîteva situaţii concrete!

**Exemplul 8.12** În *Exemplul 6.2* am arătat cum un semiautomat finit poate recunoaşte un limbaj regulat (poate recunoaşte apartenenţa unei secvenţe regulate de $n$ simboluri la o mulţime de şiruri). Aceeaşi problemă poate fi rezolvată şi cu echivalentul combinaţional al unui semiautomat (vezi subcapitolul 5.9). Există şi o a treia soluţie oferită de un *ROM* cu un număr de intrări proporţional cu $n$. În cazul *(a)* al semiautomatului finit scriem

$$S(n) = O(1) \text{ şi } T(n) = O(n)$$

pentru complexitatea circuitului care rezolvă analiza unui şir de $n$ simboluri în $T(n)$. În cazul *(b)* al reţelei combinaţionale asociate am obţinut

$$S(n) = O(n) \text{ şi } T(n) = O(\log n)$$

pentru rezolvarea aceleiaşi probleme. Iar în cazul *(c)* al unei soluţii cu *ROM* vom scrie

$$S(n) = O(2^n) \text{ şi } T(n) = O(1).$$

Odată cu creşterea vitezei de rezolvare a problemei (scăderea lui $T(n)$), apare o creştere a complexităţii structurii implicate; dar se mai produce un efect: complexitatea creşte mult mai rapid. Altfel spus, odată cu scăderea lui $T(n)$ creşte produsul $S(n)T(n)$. Într-adevăr apar următoarele corespondenţe:



$$(a)\ T(n) = O(n) \rightarrow T(n)S(n) = O(n),$$
$$(b)\ T(n) = O(\log n) \rightarrow T(n)S(n) = O(n \log n),$$
$$(c)\ T(n) = O(1) \rightarrow T(n)S(n) = O(2^n).$$

Preţul plătit este disproporţionat de mare. În cazul *(c)* el nici nu mai poate fi plătit, poate nici măcar cu tehnologiile mileniului trei. Merită, poate, să facem efortul de a ajunge la soluţia *(b)*, dar cea mai eficientă rămîne în continuare prima soluţie, cea oferită de semiautomat (mai ales dacă ţinem cont şi de comentariul anterior). □

**Exemplul 8.13** Merită să analizăm din aceeaşi perspectivă şi unele soluţii date reţelelor pentru calculul prefixelor (*RCP*).

*(a)* În *Exemplul 8.4* reţeaua este caracterizată prin

$$S(n) = c_1(n \log_2 n - (n-1)),\ T(n) = \log_2 n.$$

*(b) Propoziţia 8.2* introduce o reţea caracterizată prin

$$S(n) = c_1(2n - 2 - \log_2 n),\ T(n) = 2 \log_2 n.$$

*(c)* Prin *Propoziţia 8.1* este propusă o reţea cu

$$S(n) = c_2 \frac{1}{2}(n + 2)(n - 1),\ T(n) = 1.$$

Rezultă şi în acest caz că odată cu scăderea lui $T(n)$, creşte produsul $T(n)S(n)$. □

**Exemplul 8.14** Rotorul (un caz particular al reţelei de permutare) studiat în paragraful 3.5.3, a fost realizat în două variante. Prima cu

$$S(n) = O(n^2 \log n) \text{ şi } T(n) = O(1),$$

iar cea de a doua cu

$$S(n) = O(n \log n) \text{ şi } T(n) = O(\log n).$$

Observăm scăderea produsului $S(n)T(n)$ de la $O(n^2 \log n)$ la $O(n \log^2 n)$, odată cu creşterea lui $T(n)$ de la $O(1)$ la $O(\log n)$. □

**Exemplul 8.15** Putem aduce în sprijinul aceleiaşi idei şi comentariile ce au fost făcute asupra *Propoziţiei 5.13*, privind accentuarea caracterului secvenţial al evoluţiei unui $AS^3$ (paragraful 5.7.1). □

**Exemplul 8.16** Un decodificator *(DEC)* cu adîncime constantă are complexitatea $S_{DEC}(n) = O(n\,2^n)$, unde $n$ reprezintă numărul de intrări. Structura lui se poate minimiza aplicînd algoritmul sugerat de schema din figura 8.33, unde fiecare ieşire este calculată cu un *AND* cu două intrări comandate de ieşirile a două *DEC* cu $n/2$ intrări. Rezultă o complexitate:

$$S_{DEC}(n) = 2 \cdot 2^n + 2 \cdot S_{DEC}(n/2).$$

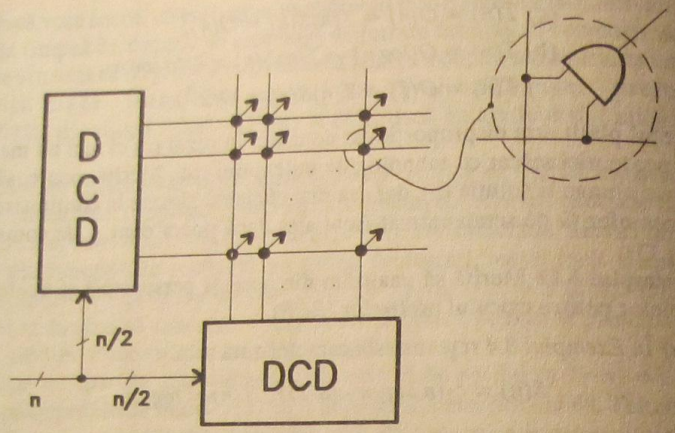



**Figura 8.33**

Aplicînd, în continuare, aceeaşi regulă şi pentru *DEC* cu *n*/2 intrări, rezultă:

$$S_{DEC}(n) = O(2^n),$$
$$D_{DEC}(n) = O(\log n).$$

Deci, acceptînd o adîncime logaritmică produsul $S_{DEC}(n)D_{DEC}(n)$ scade de la $O(n\,2^n)$ la $O(2^n \log n)$.

Şi în acest caz scăderea vitezei este acoperitor compensată prin scăderea complexităţii structurii circuitului. □

Am putea risca, în consecinţă, o ultimă propoziţie.

**Propoziţia 8.10** Pentru o funcţie dată, implementările ce presupun scăderea lui $T(n)$ sînt caracterizate prin creşterea produsului $T(n)S(n)$. □

Sînt mari şansele ca demonstrarea acestei propoziţii să poată fi realizată.



9

# ANEXE

## 9.1 Notaţii neexplicate în text

- $|X|$ - cardinalul mulţimii $X$.
- $P^*(X)$ - mulţimea părţilor nevide a mulţimii $X$.
- $\lfloor N \rfloor$ - cel mai mare întreg mai mic sau egal cu numărul $N$;
  $\lceil N \rceil$ - cel mai mic întreg mai mare decît numărul $N$.
- $f(n) = O(g(n))$ dacă există o constantă $c$, astfel încît pentru $n > n_0$ este îndeplinită condiţia ca $f(n) < c \cdot g(n)$.

**Exemplul 9.1**

$$f(n) = an^2 + bn + c = O(n^2). \square$$

## 9.2 Parametrii porţilor logice elementare

### 9.2.1 Caracteristica de transfer

Caracteristica de transfer a unei porţi elementare poate fi dată sub forma mărimilor limită, întabelate sau sub formă grafică.

Mărimile limită sînt următoarele:

- $V_{IL}$, este tensiunea de intrare recunoscută ca avînd valoarea zero logic (sau fals);
- $V_{IH}$, este tensiunea de intrare pe care circuitul o interpretează ca unu logic (sau adevărat);
- $V_{OL}$, este nivelul de tensiune la ieşire prin care circuitul generează zero logic;
- $V_{OH}$, este nivelul de tensiune la ieşire prin care circuitul generează unu logic;
- ($V_T$, este tensiunea de prag).

Aceşti parametri depind de: temperatura ambiantă ($T_A$), valoarea sursei de alimentare ($V_{CC}$) şi de fan-out-ul din schemă ($N$).

**Exemplul 9.2** În tabelele 9.1 şi 9.2 sînt ilustrate dependenţele de $T_A$ şi $V_{CC}$, pentru $N = 10$, în cazul circuitelor standard *TTL* (după lucrarea [Morris '71]). □

Caracteristica de transfer în tensiune ilustrează mai complet comportamentul circuitului logic sub forma dependenţei $V_O = f(V_I)$.

| $T_A$ | -55°C | 0°C | 25°C | 70°C | 125°C |
|---|---|---|---|---|---|
| $V_{OH}$ | 3 V | 3,1 V | 3,25 V | 3,3 V | 3,5 V |
| $V_{OL}$ | 0,25 V | 0,29 V | 0,3 V | 0,31 V | 0,32 V |
| $V_T$ | 1,5 V | 1,4 V | 1,3 V | 1,2 V | 1 V |

$V_{CC} = 5$ V, N = 10

**Tabelul 9.1**

| $V_{CC}$ | 4,5 V | 4,75 V | 5 V | 5,25 V | 5,5 V |
|---|---|---|---|---|---|
| $V_{OH}$ | 2,6 V | 2,85 V | 3,25 V | 3,35 V | 3,55 V |
| $V_{OL}$ | 0,33 V | 0,32 V | 0,3 V | 0,3 V | 0,3 V |
| $V_T$ | 1.28 V | 1,29 V | 1,3 V | 1,32 V | 1,35 V |

$T_A = 25$ °C, N = 10

**Tabelul 9.2**

**Exemplul 9.3** În figura 9.1 sînt date caracteristicile de transfer ale inversoarelor realizate în tehnologie *CMOS (a)* şi *TTL (b)*. □



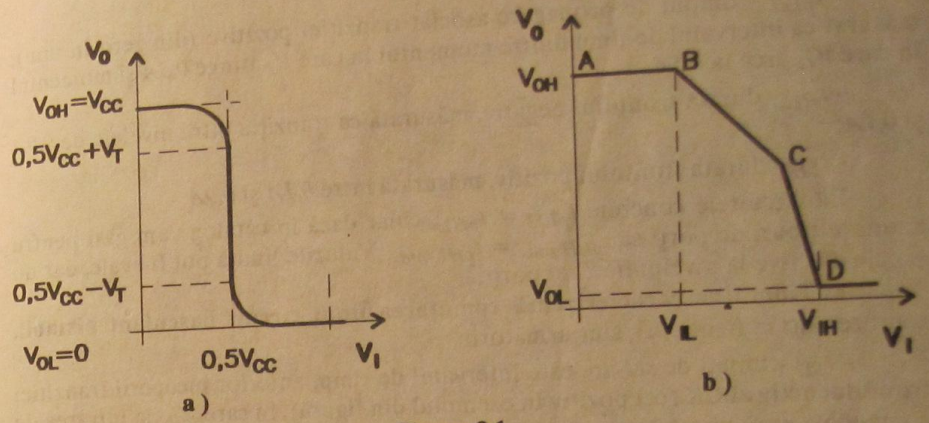



**Figura 9.1**

### 9.2.2 Timpii de comutaţie

• **Timpii de comutaţie ce caracterizează funcţionarea unei porţi logice** sînt, conform reprezentării din figura 9.2, următorii:

- $t_{pHL}$ - timpul de propagare la tranziţia din unu în zero (high to low), măsurat ca intervalul de timp dintre momentul în care $V_I$ atinge *0,5A* şi $V_O$ coboară la *0,5A*; unde *A* este "amplitudinea" impulsului;

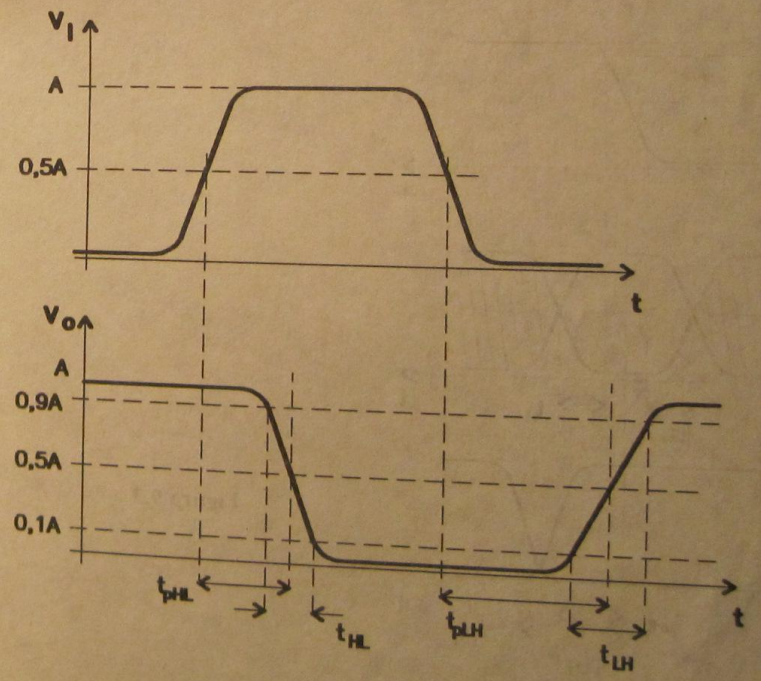



**Figura 9.2**

- $t_{pLH}$ - timpul de propagare asociat tranziţiei pozitive (din zero în unu), măsurat ca intervalul de timp dintre momentul în care $V_I$ atinge *0,5A* şi momentul în care $V_O$ urcă la *0,5A*;

- $t_{HL}$ - durata frontului negativ, măsurată ca tranziţie între nivelele de *0,9A* şi *0,1A*;

- $t_{LH}$ - durata frontului pozitiv, măsurată între *0,1A* şi *0,9A*.

La circuitele concrete $t_{pLH} \neq t_{pHL}$, chiar dacă în catalog vom găsi pentru anumite tipuri de porţi că $t_{pLH\,max} = t_{pHL\,max}$. Valorile limită pot fi egale, dar nu şi cele efective la nivelul fiecărei porţi.

• **Timpii ce caracterizează comutarea unui circuit basculant bistabil**, reprezentaţi în figura 9.3, sînt următorii:

- $t_{SU}$ -timpul de *set-up*, este intervalul de timp, anterior începerii tranziţiei frontului activ al *CK* (cel pozitiv în exemplul din figură), în care $V_{IN}$ la intrarea de comandă a unui bistabil trebuie să fie stabil;

- $t_H$ - timpul de *hold*, este intervalul de timp, posterior încheierii tranziţiei ceasului, în care intrarea $V_{IN}$ mai trebuie menţinută stabilă;

- $t_{pHL}$ - timpul de propagare faţă de *CK* la comutarea ieşirii ($V_{OUT}$) în zero; se măsoară din momentul în care *CK* atinge *0,5A* iar $V_{OUT}$ coboară la *0,5A*;

- $t_{pLH}$ - timpul de propagare faţă de *CK* la comutarea ieşirii în unu.

Şi în acest caz se definesc timpii de front $t_{LH}$ şi $t_{HL}$ identici cu cei de la porţi.

Deci frontul ceasului asigură comutarea unui bistabil într-un "moment" aproximat prin $t_{SU} + t_{LH} + t_H$ (unde prin $t_{LH}$ am notat timpul de comutare al *CK*).

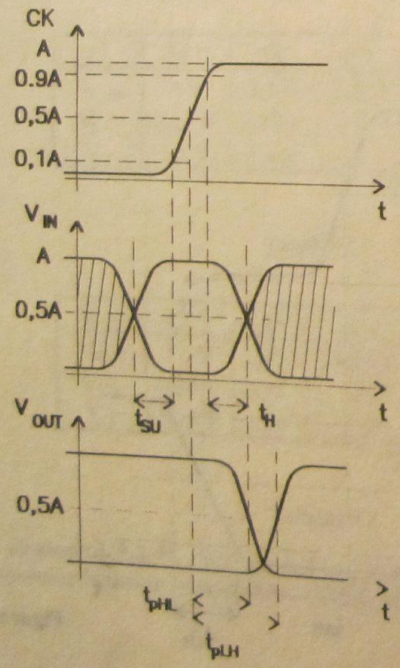



**Figura 9.3**



### 9.2.3 Marginea de zgomot

Pe traseul ce conectează ieşirea unui circuit la intrarea altuia se transmite un semnal peste care se poate suprapune şi un altul asimilabil cu un zgomot. Cît de mare poate fi acest semnal pentru a nu perturba funcţionarea sistemului?

Parametrul ce defineşte imunitatea la zgomot este denumit *margine de zgomot* şi se notează cu $NM_H$ şi $NM_L$ (*n*oise *m*argin), pentru nivelul înalt *(H)* şi cel coborît *(L)*.

**Definiţia 9.1** Marginea de zgomot este dată de diferenţa dintre gama de intrare permisă şi gama de ieşire garantată la cuplarea dintre două porţi logice (figura 9.4), conform relaţiilor:

$$NM_H = V_{OH\,min} - V_{IH\,min}$$
$$NM_L = V_{IL\,max} - V_{OL\,max}. \quad \square$$

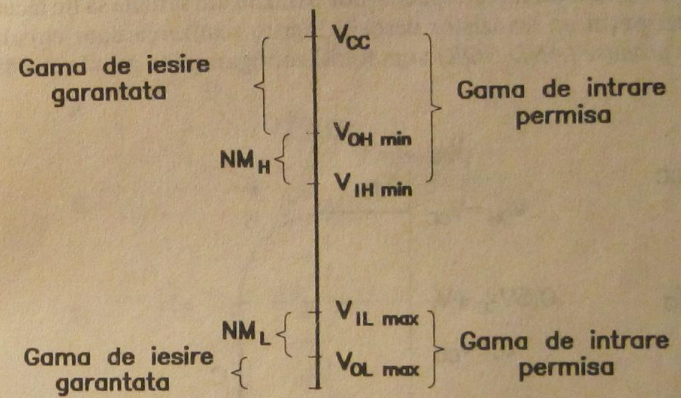



**Figura 9.4**

## 9.3 Porţi elementare

Prezentăm sumar numai două familii de circuite integrate: familia *CMOS* (*C*omplementary - symmetry *M*etal - *O*xide *S*emiconductor) şi familia *TTL* (*T*ransistor - *T*ransistor *L*ogic).

### 9.3.1 Circuite CMOS

• **Structura fundamentală** este constituită de *inversorul CMOS* reprezentat în figura 9.5*a*. Funcţia de transfer asociată este redată în figura 9.5*b*, unde sînt puse în evidenţă cinci zone distincte:

- zona *A* - în care tranzistorul $Q_1$ este blocat iar $Q_2$ în conducţie în zona ohmică a caracteristicilor;

- zona *B* - în care, fiind depăşită tensiunea de prag a tranzistorului $Q_1$, acesta intră în zona de saturaţie, $Q_2$ rămînînd în zona ohmică;

- zona *C* - atît $Q_1$ cît şi $Q_2$ sînt în zona de saturaţie a caracteristicilor;

- zona *D* - $Q_1$ ajunge în zona rezistivă (ohmică) iar $Q_2$ este în continuare în saturaţie;

- zona *E* - $Q_1$ este în zona rezistivă iar $Q_2$ este blocat.

În zonele extreme, *A* şi *E*, prin circuit curentul este nul. $I_D$ va fi maxim pentru $V_I = 0{,}5V_{CC}$, conform reprezentării din figura 9.5*c*. Consumul intern al unui astfel de circuit va depinde de frecvenţa comutărilor.

• **Porţile NAND şi NOR** realizate în tehnologie *CMOS* au configuraţia din figurile 9.6*a*, respectiv 9.6*b*. Fiecare intrare presupune o pereche de tranzistoare, astfel încît încărcarea sau descărcarea capacităţilor parazite din sarcină să fie făcută întotdeauna prin cel puţin un tranzistor deschis. Pentru realizarea unor circuite asociate *sumelor de produse (AND NOR)* vom folosi configuraţii de genul celei din figura 9.7.

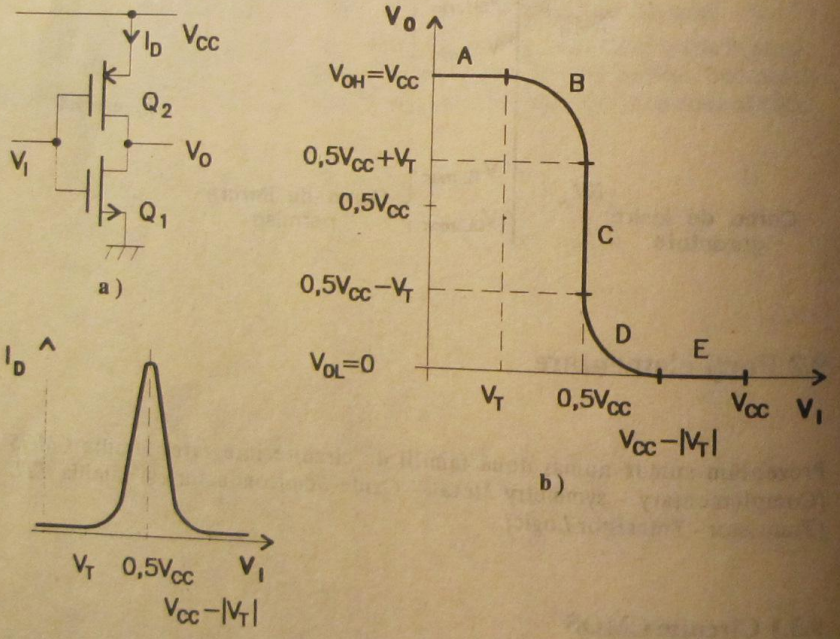



**Figura 9.5**



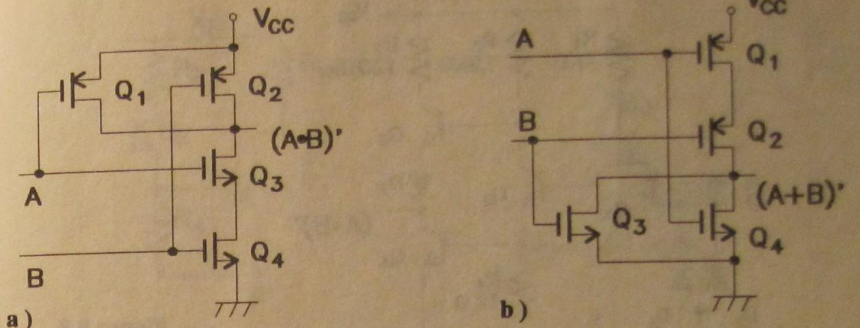



**Figura 9.6**

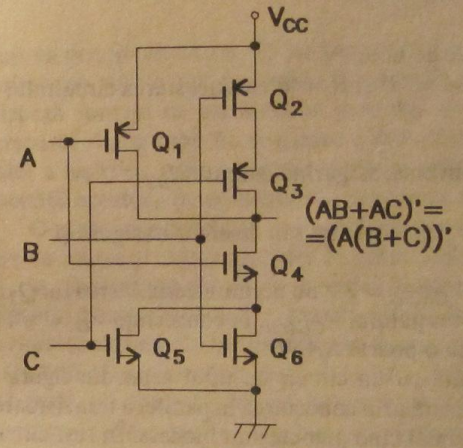



**Figura 9.7**

## 9.3.2 Circuite TTL

• **Circuitul de bază** al familiei *TTL* îl constituie poarta NAND cu două intrări reprezentată în figura 9.8.

Dacă $A = B = "0" \approx 0{,}2\,V$ atunci:

- $Q_1$ este saturat de curentul prin $R_1$ la un curent de colector practic nul, deci, $V_{B2} = 0{,}2\,V$ şi în consecinţă:

- $Q_2$ este blocat iar curentul prin $R_4$ fiind nul, $V_{B4}$ este, de asemenea nul, rezultînd:

- $Q_4$ blocat;

- $Q_3$ este deschis de curentul prin $R_2$, dar valoarea mică a rezistenţei $R_3$, la un curent de ieşire normal (sub *1 mA*), păstrează tranzistorul în regiunea activă normală.

Deci, la ieşire avem

$$V_O = V_{CC} - (I_{B3}R_2 + V_{BE3} + V_D) \approx 3{,}8\,V$$

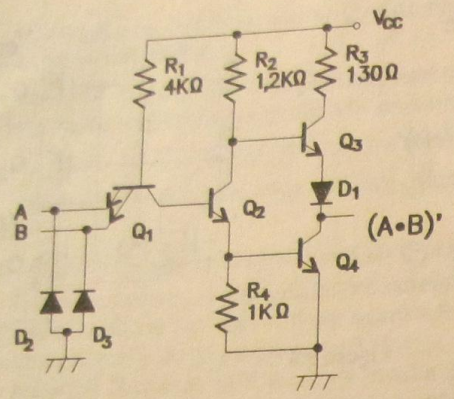



**Figura 9.8**

ceea ce corespunde semnalului logic de unu.

Dacă, $A$ = "*0*" şi $B$ = "*1*" sau $A$ = "*1*" şi $B$ = "*0*", atunci starea circuitului este aceeaşi.

Dacă, $A = B$ = "*1*", atunci:

- $Q_1$ este în regiunea activă inversă, asigurînd curentul $I_{B2}$

astfel că:

- $Q_2$ este saturat puternic iar prin curentul său de emitor asigură şi
- $Q_4$ în starea de saturaţie.

Potenţialul $V_{B3} = V_{BE4sat} + V_{CE2sat} \approx 1\,V$ nu permite deschiderea lui $Q_3$ care este blocat datorită diodei $D_1$ şi potenţialului $V_{CE4sat}$. În consecinţă $V_O$ = "*0*".

Deci circuitul funcţionează ca o poartă *NAND*.

• **Funcţia NOR** este realizată cu un circuit de tipul celui din figura 9.9. Funcţia logică propriuzisă este realizată prin conectarea în paralel a tranzistoarelor $Q_2$ şi $Q_3$, unu logic obţinîndu-se numai cînd ambele sînt blocate. În rest circuitul funcţionează identic. Pentru circuite de tip *AND-NOR* este suficient ca tranzistoarele $Q_1$ şi $Q_4$ să fie dotate cu un număr corespunzător de emitoare.

• **Circuitul cu ieşire open collector** se obţine eliminînd de la ieşirea circuitului *TTL* rezistenţa $R_3$, împreună cu tranzistorul $Q_3$ şi dioda $D_1$. Circuitul va putea genera, în continuare un bun "*0*", dar pentru "*1*" va fi necesară cuplarea în exterior a unei *rezistenţe de pull-up*, conform reprezentării din figura 9.10*a*. Acest

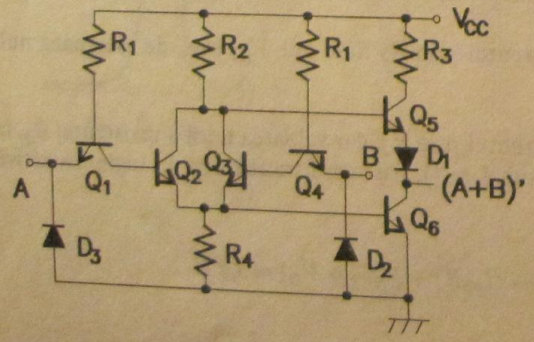
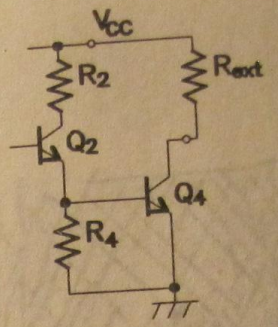
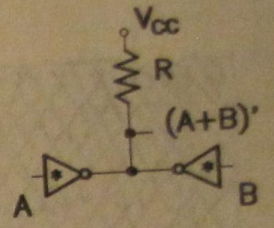



**Figura 9.9**






a)



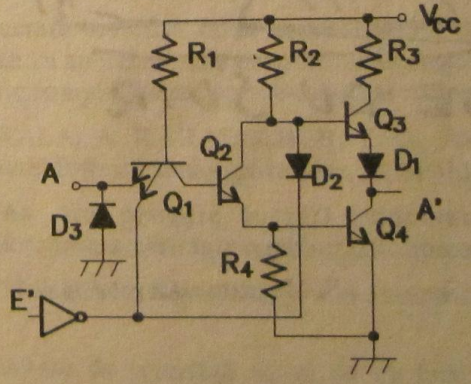

b)

**Figura 9.10**

circuit va genera autoritar "*0*" iar pentru "*1*" va fi permisiv. Vom nota cu o steluţă circuitele ce au o astfel de ieşire. În figura 9.10*b* este reprezentat un circuit ce-şi realizează funcţia ca un efect al modului de conectare al ieşirilor celor două inversoare. Acest mod de realizare a circuitului de ieşire permite conectarea în paralel a ieşirilor unor structuri digitale, asigurînd astfel posibilitatea extinderii capacităţii acestora (spre exemplu, extinderea capacităţii *ROM*-urilor).

Generarea electrică a semnalului "*1*" se face în condiţii precare, fapt pentru care s-au impus şi alte modalităţi de a face ieşirile neimperative.

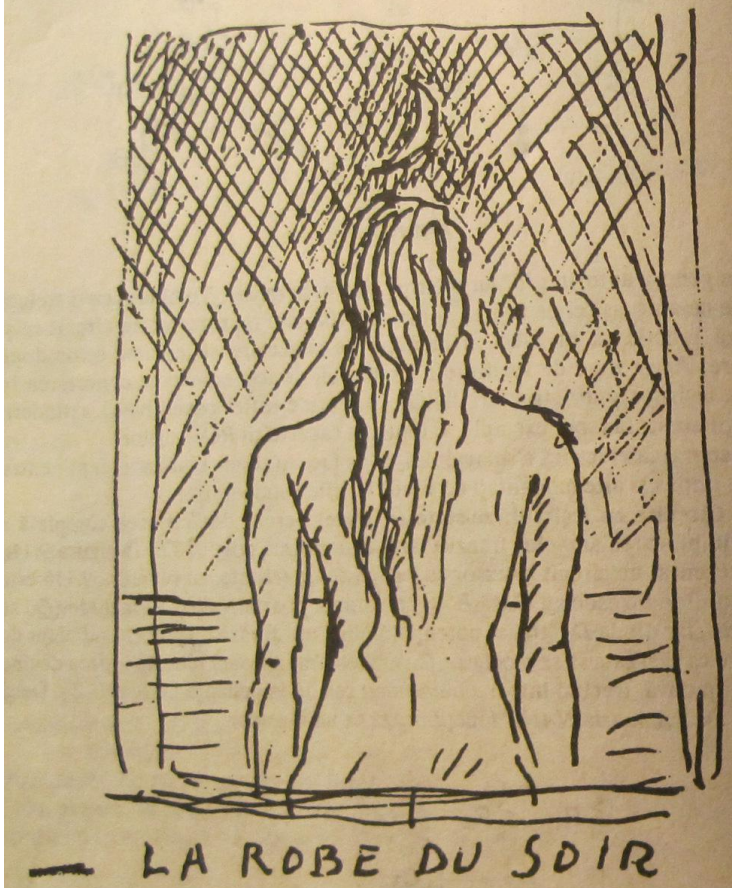
• **Circuite cu ieşiri three-state (tristate)** permit dezactivarea completă a ieşirii prin blocarea ambelor tranzistoare de la ieşirea porţii *TTL*. În figura 9.11*a* este reprezentat un circuit inversor cu ieşirea de tip tristate, iar în figura 9.11*b* este dat simbolul logic asociat. Dacă $E'$ = "*1*", atunci poarta *NAND* se închide ($Q_4$ se blochează), iar dioda $D_2$ "trage" potenţialul din baza tranzistorului $Q_3$ suficient de jos pentru ca şi acesta să se blocheze. Cu ambele tranzistoare blocate ieşirea devine complet inactivă, trecînd într-o *a treia stare*, cea de impedanţă mare (*Hi-Z*). Dacă $E'$ = "*0*", atunci poarta *NAND* funcţionează ca un inversor.




a)

b)

**Figura 9.11**




R. Magritte : *La robe du soir*

# 10

# COMENTARII BIBLIOGRAFICE

Rolul acestor comentarii bibliografice este în primul rînd istoric. Sînt prezentate lucrările de referinţă, originare, ale domeniului. De asemenea se urmăreşte recomandarea unor lecturi suplimentare, didactice, prin care studentul îşi poate completa cunoştinţele dobîndite după asimilarea cursului. Şi, nu în ultimă instanţă, autorul îşi prezintă lucrările mai importante în care a publicat aspectele originale ce apar în acest curs. O bibliografie consistentă a domeniului ar reprezenta ea însăşi o carte şi nu este utilă celui ce, ca începător, parcurge această lucrare.

ARBIB, A. M., *Theories of Abstract Automata,* Prentice-Hall Inc., Englewood Cliffs, N.J., 1969.

Lucrare teoretică fundamentală pentru conceptul de automat abstract şi pentru aplicaţiile sale în domeniul limbajelor şi calculului. Tratarea formală este completată cu aspecte calitative limpezitoare.

AGRAWALA, A. K., RAUSCHER, T. G., *Foundations of Microprogramming; Architecture, Software and Applications*, ACM Monograph, N.Y.: ACM Press, 1976.

Una din primele lucrări importante care prezintă domeniul microprogramării într-o monografie aproape completă.

BLAKESLEE, T. R., *Digital Design with Standard MSI and LSI*, John Wiley & Sons, 1979.

Lucrare de referinţă scrisă de un bun practician. Prezintă tehnici de proiectare logică şi electrică cu un accent deosebit pus pe problemele ridicate de realizarea concretă şi pe aspectele extrainginereşti ale

domeniului. Nivelul de complexitate abordat este cel al circuitelor *MSI* (*m*edium *s*cale *i*ntegration) şi *LSI* (*l*arge *s*cale *i*ntegration). Traducerea în limba română (Ed.Tehnică, 1988) beneficiază de o postfaţă, scrisă de M. Bodea şi Gh. Ştefan, în care sînt aduse la zi o parte din problemele tratate în lucrarea originală.

BOOLE, G. , *Investigations of the Laws of Thought*, London, Eng.: Walton, 1854.

BOOLE, G. , *The Mathematical Analysis of Logic*, Cambridge, Eng., 1847.

Texte fundamentale pentru formalizarea logicii clasice. Au permis dezvoltarea unei algebre logice (booleene) ce descrie procesele fundamentale dintr-un sistem digital.

CALUDE, C. , *Complexitatea calculului. Aspecte calitative*. Ed. Ştiinţifică şi Enciclopedică, Bucureşti, 1982.

Depăşeşte mult domeniul circuitelor digitale. Poate fi consultată pentru prezentarea limbajului LOOP şi a aplicaţiilor acestuia.

CLARE, C. , *Designing Logic Systems Using State Machine*, Mc Graw-Hill, Inc., 1972.

Lucrare de referinţă pentru tehnicile de proiectare a automatelor finite elementare. Completează bine lucrarea lui T. Blakeslee. Tratează sistematic probleme de cuplare a automatelor finite într-un sistem digital mai amplu.

DRĂGĂNESCU, M., ŞTEFAN, GH., BURILEANU, C., *Electronica funcţională*, Ed. Tehnică, Bucureşti, 1991.

Domeniul abordat de această carte este mult extins peste cel al sistemelor digitale. Prezintă interes direct numai tratarea unor probleme legate de *informaţie* şi de *abordarea structurală* în electronică.

ENESCU, GH., *Logică şi adevăr*, Ed. Politică, Bucureşti, 1967.

O carte scrisă de un logician, utilă inginerilor care doresc să înţeleagă logica şi dincolo de purul ei formalism.

GLUSHKOV, V. M., LETICHEVSKII, A. A., *Theory of Algorithms and Discrete Processors*, în "Advances in Information System Sciences", Vol.1, Ed. Julius Tou, 1964.

GLUSHKOV, V. M., *Introduction to Cybernetics*, Academic Press, N.Y., 1966.

Glushkov introduce conceptul de *maşină algoritmică*. Această structură rezultă prin interconectarea în buclă a unui automat de comandă cu un automat operaţional (efector). Modelul lui Glushkov este cel al procesorului, ca sistem de ordinul trei în accepţiunea pe care am dat-o în cursul de faţă.

HABIB, S. ed., *Microprogramming and Firmware Engineering Methods*, Van Nostrand Reinhold, N.Y., 1988.



Cartea cuprinde un istoric al acestei tehnici precum şi metode şi tehnici de ultimă oră în domeniu.

HAZNEDAR, H., *Digital Microelectronics*, The Benjamin/Cummings Pub. Comp., Inc., 1991.

Conţine o prezentare la zi a circuitelor de comutaţie integrate. Pornind de la principiile ştiinţei semiconductoarelor sînt studiate circuitele elementare (porţi şi bistabili) realizate în principalele tehnologii (TTL, MOS, CMOS, I²L, ECL), memoriile, circuitele sample-and-hold şi cele de conversie A-D şi D-A. Cartea se încheie cu un capitol destinat efectelor parazite în sistemele digitale. Reprezintă un material ce completează foarte bine lucrarea de faţă.

HUSSON, S. S., *Microprogramming Principles and Practices*, Englewood Cliffs, N.J., Prentice Hall, 1970.

Prima carte scrisă despre microprogramare. Conţine, în primul rînd, descrierea principalelor procesoare microprogramate produse pînă la apariţia sa.

MEAD, C., CONWAY, L., *Introduction to VLSI Systems*, Addison-Wesley Pub. Comp., 1980.

Este o primă expunere sistematică a sistemelor digitale din categoria VLSI. Prezentarea teoretică este exemplificată prin proiecte concrete. În carte apare şi un text de referinţă pentru sistemele sistolice scris de H. T. Kung şi C. Leiserson.

MEALY, G. W., *A Method of Synthetising Sequential Circuits*, BSTJ, 34, 1045- 1079, 1955.

Textul prezintă modelul automatului finit Mealy.

MOISIL, G. C., *Teoria algebrică a schemelor cu contacte şi relee*, Ed. Tehnică, Bucureşti, 1965.

Reprezintă contribuţia şcolii româneşti la dezvoltarea teoriei ce fundamentează realizarea sistemelor digitale.

MOORE, E. F., *Gedanken - Experiments on Sequential Machines*, Automata Studies (Annals of Mathematics Studies, No.34), Princeton Univ. Press, 1956.

Textul este important pentru prezentarea modelului Moore al automatelor finite.

MORRIS, L. R., MILLER, J. L. ed., *Designing with TTL Integrated Circuits*, Texas Instruments Inc., 1971.

Un prim text de referinţă pentru circuitele integrate digitale. A fost tradus în limba română (Ed. Tehnică, 1974). Este prezentată familia de circuite TTL de mică complexitate (SSI - smal scale integration) prin parametrii electrici şi cei ce caracterizează timingul acestor circuite. Conţine şi aplicaţii standard.

NOVICOV, P. S., *Elemente de logică matematică*, Ed. Ştiinţifică, Bucureşti, 1966.

Lucrare pur teoretică ce prezintă sistematic algebra booleană.

RUSSEL, B., WHITEHEAD, A. N., *Principia Mathematica*, 3 vol., Cambridge, Eng., The University Press, 1910-1913.

Lucrare fundamentală pentru matematică în general şi pentru dezvoltarea logicii matematice, în special. Apare din efortul de a dezvolta formal domeniul matematicii în contextul în care fundamentele acesteia păreau şubrezite de apariţia şi proliferarea paradoxurilor logico-matematice. Momentul istoric în care apare este marcat de încercările de a dovedi caracterul *complet* şi *necontradictoriu* al teoriilor matematice. Criza sistemelor formale se va răsfrînge şi asupra unor probleme esenţiale din teoria sistemelor digitale de calcul (problema opririi unui program pe o maşină dată).

SAVAGE, J. E., *The Complexity of Computing*, R. E. Krieger Pub. Comp., 1987.

Este o carte de referinţă pentru complexitatea circuitelor digitale şi a sistemelor de calcul. Conţine teoremele fundamentale ale domeniului. (O demonstrare în cazul general a teoremei lui Spira va putea fi găsită în această lucrare.)

SHANNON, C., *A Symbolic Analysis of Relay and Switching Circuits*, în Transactions of the American Institute of Electrical Engineers, vol.57, p.713-723, 1938.

Articol de importanţă istorică pentru domeniul circuitelor logice. Este primul text în care se arată cum se poate materializa algebra booleană. După ce G. Boole formalizase algebric logica aristotelică, C.Shannon oferă o soluţie fizică care va deschide calea conceperii şi realizării calculatoarelor.

ŞTEFAN, G., *Structurări neechilibrate în sisteme de prelucrare a informaţiei*, Comunicare la sesiunea ştiinţifică "Inteligenţa artificială şi robotica" organizată de Academia RSR în oct. 1981, publicată în vol. *Inteligenţa artificială şi robotica*, Ed. Academiei RSR, Bucureşti, 1983.

În această lucrare este publicat pentru prima oară modelul (propus în teza de doctorat, 1979) structurării sistemelor digitale, expus în capitolul doi al prezentei cărţi. Tot aici este evidenţiat şi mecanismul de structurare neechilibrată.

STEFAN, G., DRĂGHICI, I., MUREŞAN, T., BARBU, E., *Circuite integrate digitale*, Ed. Didactică şi Pedagogică, Bucureşti, 1983.

Este prima variantă a cursului de circuite integrate digitale realizată conform principiului de creştere structurală sub formă de ordine. Principiul este reexpus şi amplu exemplificat.

STEFAN, G., BISTRICEANU, V., PĂUN, A., *Către un mod natural de implementare a LISP-ului*, Comunicare la al doilea Simpozion Naţional de Inteligenţă Artificială, organizat de Academia RSR în sept. 1985, publicată în vol. *Sisteme de inteligenţă artificială*, Ed. Academiei Române, Bucureşti, 1991.

Lucrare în care este expusă pentru prima dată ideea *memoriei conexe*.

STEFAN, G., *Funcţie şi structură în sistemele digitale*, Ed. Academiei Române, Bucureşti, 1991.

Pe lîngă o serie de aspecte clasice, de uz didactic, apare o expunere succintă a ideilor legate de calculul paralel. Recomandăm lucrarea pentru demonstrarea corespondenţei ce există între clasificarea limbajelor pe tipuri, făcută de N. Chomsky, şi clasificarea sistemelor digitale, ce recunosc sau generează aceste limbaje, în ordine. Punerea în corespondenţă a tipurilor de limbaje cu ordinul maşinilor fizice asociate vine să valideze suplimentar ambele clasificări şi mecanisme. Se evidenţiază şi o corespondenţă spectaculoasă între funcţia şi structura unui sistem digital.

STEFAN, G., BISTRICEANU, V. . *Circuite integrate digitale. Probleme. Proiectare*, Ed. Didactică şi Pedagogică, Bucureşti, 1992.

Constituie un material intim legat de acest curs. Conţine numai probleme rezolvate şi aplicaţii soluţionate la nivelul proiectării de sistem. Conţine următoarele capitole: Circuite combinaţionale, Memorii, Automate finite, Sisteme microprogramate, Aplicaţii. Sînt abordate şi probleme de implementare concretă ce depăşesc expunerea pe care o facem în această carte.

ULLMAN, J. D., *Computational aspects of VLSI*, Computer Science Press, 1984.

Cartea discută complexitatea structurilor VLSI prin ceea ce acestea au specific. Tratarea şi rezultatele obţinute din analiza teoretică a complexităţii circuitelor sînt aici completate cu aspectele legate de realizarea concretă în tehnologii VLSI. Lucrarea este redactată la un nivel teoretic ridicat.

VENN, J., *Symbolic Logic*, London, Eng.: Macmillan & Co., 1894.

Lucrare clasică în care s-au introdus celebrele diagrame.

WESTE, N., ESHRAGHIAN, K., *Principles of CMOS VLSI Design*, Addison-Wesley, Reading, Mass., 1985.

Prezintă filosofia sistemelor digitale VLSI realizate în tehnologia CMOS. Completează lucrarea clasică a lui Mead şi Conway care avea în vedere numai tehnologia NMOS.

WILKES, M. V., *The Best Way to Design an Automatic Calculating Machine*, Report of the Manchester University Computer Inaugural Conference, Manchester University, Manchester, England, July, 1951.

Este lucrarea în care se introduce conceptul de microprogramare. Odată cu aceasta sînt propuse ideile de transfer între registre, de microcod, de memorie de control, de structură logică a microprogramului.

WHITE, D. E., *Bit-Slice Design: Controlers and ALUs*, Garland STPM Press, N.Y., 1981.

Cartea prezintă, într-o manieră accesibilă, aplicaţii de structuri microprogramate centrate pe circuitele din seria 2900. În această serie structura de bază o constituie un RALU structurat pe cuvinte de patru biţi, extensibil. Circuitul are 16 registre şi o unitate logico-aritmetică cu opt funcţii.

# Abstract

The main idea of this book starts from a structuring principle of digital integrated circuits, assuming growing mechanisms through: serial, parallel, serial-parallel extensions and the loop connected configuration (in figure 2.4). If only the extensions are used, we remain in the same class (order) of functions and structures. When a loop is introduced, a functional growth is obtained and the order of the digital system is incremented.

The circuits that perform combinational functions are loopless and make up the zero order systems (*0OS*). The *NAND* gate is the basic element in *0OS*, the demultiplexer is an example of parallel extension in *0OS* and the multiplexer is the serial-parallel extension.

The first order systems (*1OS*) have memory functions. The basic structure is the elementary latch (figure 4.6), the serial extension is the master-slave structure, the parallel extension is the random access memory and the serial-parallel extension is the register.

When we introduce another loop, the second order systems (*2OS*) are generated. The basic structure in this order is the finite automaton. In the middle of the simplest automata we mention *T* and *JK* flip-flops. One functionally oriented automaton is the counter. The most complex automata studied are the *RALU* (registers with *ALU*) and the *CROM* (controller with *ROM*).

The processor is the illustrative structure of the third order structures (*3OS*). It is obtained by a loop connecting two automata: a *RALU* and a *CROM*, in most cases. In the same order we may introduce finite automata centered on "*JK* registers" or binary synchron presetable counters, as optimal solutions.

The transition from an order to the next brings about the increased autonomy of the system. Each order is associated with a function level, as follows:

– *0OS* : combinational functions,

– *1OS* : memory functions,

– *2OS* : finite automata,

– *3OS* : processing structures,

– *4OS* : computer systems (not developed in this book).

Any function performed in *NOS* is also performed in *(N+1)OS*.

At the level of the *2OS* (in *RALU*) information structures turn up and, beginning with *3OS*, monopolize step by step, the functional characterization. Under these conditions, our approach stops at *3OS*, because of its structural nature.

The last section of the book is devoted to complex systems, expanded in parallel networks. The size and time in combinational networks, networks with memory and automata networks is analyzed. Emphasis is laid on the relation between time *(T(n))* and size *(S(n))* for some algorithms and implementations.

The final suggestion is that , if a solution, at any level, presupposes the decrease of *T(n)*, then the product *S(n)T(n)* increases.



# Extended contents


**1. INTRODUCTION** 19

Continuous - discontinuous. Signal - symbol. Combinational - sequential. Universal machines - functional structures. Formal - nonformal

**2. DIGITAL SYSTEM STRUCTURING** 25

Serial extention. Parallel extention. Serial-parallel extention. Loop extention. Order classification.

**3. COMBINATIONAL LOGIC CIRCUITS**

**Zero order systems** 31

3.1 Formal description of logic circuits 31

3.1.1 Variables and functions 31

3.1.2 Logical connectives 33

3.1.3 Formal logic 36

3.1.4 Logical functions minimization 40

Algebrical procedures. Veitch-Karnaugh diagrams. Incompletely defined functions. Diagrams with including variables and functions.

3.1.5 Functions defined in natural language 48

3.2 Combinational circuits implemented with gates 50

3.2.1 One output circuits 50

Propagation time. Fan-out and fan-in. Using inverting gates. Size and depth of circuits. Hazard. The avoidance of hazard.

3.2.2 Many outputs circuits 59

Full adder. Global minimization. Depth optimization. 62
3.3 Using standard circuits in implementation 62
3.3.1 Circuits implemented with multiplexers (MUX)
MUX with logical values on selected inputs. MUX with variables on inputs. MUX with functions on inputs. Comparison. 65
3.3.2 Circuits implemented with demultiplexers (DMUX)
Opportunity. Using NAND and AND. A circuit for exponential function. 68
3.3.3 Using MUX and DMUX in combinational circuits implementation
Example. The maximum size estimation. 70
3.3.4 MUX and DMUX extentions 72
3.4 Programmable structures 73
3.4.1 Read only memory (ROM)
Definition. Two-dimensional addressing. Example. ROM extention. Efficient utilization. 79
3.4.2 Programmable logic array (PLA)
Definition. Example. Logic - arithmetic alternative. Expanding PLA's. PGA. Gates array. 83
3.5 Specialized circuits 83
3.5.1 Function oriented circuits
Adder. Comparator. Priority encoder. 84
3.5.2 Circuits with selectable functions
Optimal code for selection. ALU. 87
3.5.3 Combinational shifters
The principle. Barrell shifter. Size and depth. 90
3.6 Complexity of combinational circuits 92
3.6.1 Universal logic circuit 93
3.6.2 Spira's theorem
Examples. Proof of theorem. Applications.

**4. MEMORIES**

**Second order systems** 97
4.1 The effect of the loop over a combinational logic circuit 98
4.1.1 Unstable structures 98
4.1.2 Stable structures 99
4.2 Latches 101
4.2.1 Elementary latch 101
4.2.2 Latch with clock 104
4.2.3 D (data) latch 106
4.3 Memories 107
4.3.1 Addressable latch 107
4.3.2 RAM 109
4.3.3 Dynamic RAM (DRAM) 113
4.3.4 DRAM array 120



4.4 Master-slave structures 124
4.4.1 Master-slave principle. D (delay) flip-flop 124
4.4.2 Serial register 127
4.5 Registers 129
4.5.1 Parallel register 129
4.5.2 Serial-parallel register 129
4.6 State register 130
4.7 FIFO memory 132

**5. ELEMENTARY FINITE AUTOMATA**

**Second order systems** 135
5.1 General structure of an automaton 136
5.1.1 Functional segregation 136
5.1.2 Synchronous and asynchronous automata 139
5.1.3 Unbalanced structuring of synchronous automata 142
5.2 Bistable automata 143
5.2.1 T flip-flop 143
5.2.2 JK flip-flop 146
5.2.3 Optimal structure of D flip-flop 150
5.3 Counters ($2^n$ stable automaton) 151
5.3.1 Asynchronous counters 152
5.3.2 Synchronous counters 154
5.3.3 Programmable counters and its applications 159
5.3.4 FIFO memory structure 161
5.3.5 Moebius counters 164
5.4 Formal descriptions of finite automata 166
5.4.1 Definitions. Properties 167
5.4.2 Representations of automata 170
5.4.3 Minimizations of automata 180
5.5 Designing finite automata 181
5.5.1 State assignation 181
The isomorphism in space of states. Reduced dependency. Minimal variation. Asynchronous inputs. Hazard.
5.5.2 Fundamental limits 185
The impossibility of testing many asynchronous variables in the same state. Immediate Mealy without hazard. Delayed Mealy having asynchronous inputs. Immediate Moore with asynchronous inputs and without hazard.
5.5.3 Design examples 187
Variants for states assignment to a half-automaton. Moore automata with asynchronous inputs. Flow-chart that do not allows minimal dependency codification. The Mealy automaton and its Moore equivalent. Moore without hazard. Flow-chart that do not allows minimal variation codification. Example of immediately Moore with asynchronous inputs and without hazard. Solutions

with supplementary states. Moore-Mealy translation. Maximum frequency of clock. 204
5.6 Automaton with negative feedback 204
5.6.1 Definitions. Examples 210
5.6.2 CROM (controller with ROM) 220
5.7 Automaton with structured state space ($AS^3$) 220
5.7.1 Definition. Effects
5.7.2 The configurations of half-automata with structured state space 222
5.7.3 RALU (Registers with ALU) 225
5.8 Functional oriented automata 230
5.8.1 Products accumulator 231
5.8.2 Stack memory (LIFO) 232
5.8.3 Connex memory 235
Definition. Automaton with connex memory.
5.9 Combinational complexity of finite automata 238
5.9.1 Finite automaton - combinational circuit conversion 239
5.9.2 Combinational equivalent of an half-automaton with D(n)=O(log n) 241
Example of Spira's theorem using. The complexity of formula network. The complexity of circuit network.
5.9.3 General combinational structure associate to a finite half-automaton 246
Equivalence with a multiplexer network. The introduction of a maximal negative feedback.

**6. PROCESSORS**
**Third order systems** **249**
6.1 Automata as third order systems 250
6.1.1 Automata realized with JK "registers" 251
The rule of transition diagram transformation. Implementation method. Examples. For what the CLC dimension is reduced? For what this effect not appear for T flip-flops? Minimal clock period.
6.1.2 Automata with synchron presetable counters 259
The principles. Example. Restrictions. Cycle generator for DRAM memory.
6.2 Automata with memory 266
RALU with CR flip-flops. Autonomy growing. Maximum search in a string. Segregation in the space of states.
6.3 Push-down automaton 270
Definition. Example: recognition of a language $\{a^n b^n \mid n \geq 1\}$.
6.4 Elementary processor 272
6.4.1 Structuring principle 272
Definition. Structure. Application: mean generation for the last two numbers. Variants: CROM with increment, CROM with testing the inverse condition.
6.4.2 The connection of elementary processor 279
Direct. By FIFO. To LIFO. Bidirectional to RAM. Elementary processor as functional oriented structure.
6.5 The processor
6.5.1 Processors without address bus 283
Processor connected through two FIFO. Example: arithmetic processor. Stack oriented processor. Example: expression evaluation. Limits. 284
6.5.2 Processor with address bus 292
Structure. Loop language. Machine language. How works a processor. Machine language variants.
6.6 The microprocessor 301
6.6.1 Microprocessor architecture 302
Idea of architecture. The system in which microprocessor is interconnected. RAM function. Block diagram. Machine language (instruction set).
6.6.2 How works the microprocessor 306
Instructions form. State register. Internal structure. Pseudocod description. Flow-chart description. Register for program counter.
6.6.3 The assembler language 320
Binary code. Octal/hexadecimal code. Symbolic code. Assembler language. High level language. Assembler/Compilator. Examples.
6.6.4 Programmable logic 324
Definition. Example: "bar" generator. System with histerezis.
6.6.5 Conclusions 330
6.7 Microprogrammed processors 331
6.7.1 Microprogramming languages 332
Definition. Basic microroutine. Example: sum.
6.7.2 Structural variants of microprogrammed processors 337
Testing inverse condition. Pipe-line register. The implications of pipe-line register. CROM with LIFO.
6.7.3 Dynamic microprogramming 342
Definition. The necessity of dynamic microprogramming. Load mechanisms for SRAM. Based addressing.
6.7.4 Conclusions 345
6.8 Conclusions 345

**7. CONCLUSIONS** 347
Correlation between function and structural level. Fourth order systems (4OS). Unbalanced structuring.

8. COMPLEX FUNCTIONAL NETWORKS 351
8.1 What is a complex system? 352
8.2 Combinational networks 353
8.2.1 Prefix computing networks 354
Definition. Network with D(n)=O(log n). Recursiveley defined network. Increment circuit.
8.2.2 Sorting networks 359
Definition. Example. Zero-one principle. Batcher sorter.
8.2.3 Permutation networks 364
Definition. Waksman permutator. Applications.
8.3 Networks with memory 367
8.3.1 Pipe-line networks 367
Necessity. Example. Processing time.
8.3.2 First order systolic networks 371
Basic elements. System for performing product between a vector and an array.
8.4 Automata networks 374
8.4.1 Prefix computing network as automata network 374
8.4.2 Systolic sorter 376
Elementary sorter. Sorting network. Example of operation. Balanced structure of sorter.
8.5 Final comments 379
Functional migration. VLSI area. For what only D(n)=O(log n)? Sequential pressing.

9. ANNEXES 383
9.1 Unexplained notations 383
9.2 Elementary logic gates parameters 384
9.2.1 Transfer characteristics 384
9.2.2 Switching times 385
9.2.3 Noise margin 387
9.3 Elementary gates 387
9.3.1 CMOS circuits 387
9.3.2 TTL circuits 389

10. BIBLIOGRAPHICAL COMMENTS 393





NUMELE .....

PRENUMELE .....

LOCALITATEA .....

STR ..... NR .....

BL .....SC .....ET ..... AP .....

JUDEŢUL ..... COD .....

Vă rog să-mi expediaţi prin colet poştal cu ramburs

.....exemplare din cartea .....

.....

DATA .....SEMNĂTURA .....

Vă puteţi procura această carte trimiţînd cuponul de mai sus pe adresa:

**Editura DENIX**
Str. Popa Savu , nr. 36 , Sector 1 ,
Bucureşti , România.

2-l 996

# CIRCUITE INTEGRATE DIGITALE

este o carte ce conţine aspectele fundamentale ale analizei şi proiectării sistemelor digitale. Circuitul elementar de la care porneşte demersul întreprins este **poarta logică**. Aplicînd mecanisme de structurare, riguros definite, se ajunge pînă la descrierea şi proiectarea **procesoarelor**, ce au ca reprezentanţi tipici microprocesorul şi maşina microprogramată. Capitolul final este dedicat reţelelor digitale complexe în care se declanşează **procese paralele**. În toate secţiunile ei, lucrarea abundă în exemple semnificative care ajută înţelegerea demersului teoretic şi propun soluţii practice. Stilul este accesibil, formalizarea nu este excesivă, autorul mizînd mult pe imaginaţia, fantezia şi participarea activă a cititorului. Parcurgerea lucrării presupune un minim de cunoştinţe preliminare şi un maxim de motivaţie din partea lectorului.

- Bun echilibru între prezentarea teoretică şi exemplificarea cu aplicaţii practice.
- Scurte şi riguroase introduceri în algebra Boole şi teoria automatelor.
- Expunere amplă a:
  - circuitelor combinaţionale,
  - structurilor şi funcţiilor de memorie,
  - automatelor finite,
  - procesoarelor.
- O primă prezentare sistematică a reţelelor complexe orientate funcţional.
- Analiza complexităţii însoţeşte opţiunile teoretice şi practice, pentru prima dată într-o astfel de lucrare.
- Evaluarea tendinţelor domeniului şi prezentarea critică a problematicii este prezentă în toate capitolele.
- Evidenţierea limitelor fundamentale pe care le are domeniul sistemelor digitale la toate nivelurile.

ISBN 973-95811-0-2

LEI: 2990